Estudo de Sistemas de Arquivos Estruturados em Log e uma Proposta de Implementação para o Ambiente LINUX

Edicezar Leandro Nanni

Dissertação de Mestrado

Instituto de Computação

Universidade Estadual de Campinas

Estudo de Sistemas de Arquivos Estruturados em Log e uma Proposta de Implementação para o Ambiente LINUX

Edicezar Leandro Nanni

Junho de 1998.

Banca Examinadora:

- Prof. Dr. Célio Cardoso Guimarães (Orientador)
- Prof. Dr. Maurício Ferreira Magalhães
  (Faculdade de Engenharia Elétrica e de Computação – UNICAMP)
- Prof. Dr. Luiz Eduardo Buzato
  (Instituto de Computação – UNICAMP)
- Prof. Dra. Maria Beatriz Felgar de Toledo (suplente)
  (Instituto de Computação – UNICAMP)

Estudo de Sistemas de Arquivos Estruturados em Log e uma Proposta de Implementação para o Ambiente LINUX

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Edicezar Leandro Nanni e aprovada pela Banca Examinadora.

Campinas, 15 de junho de 1998.

Prof. Dr. Célio Cardoso Guimarães
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 15 de junho de 1998 pela Banca Examinadora composta pelos Professores Doutores

Prof. Dr. Maurício Ferreira Magalhães

Prof. Dr. Luiz Eduardo Buzato

Prof. Dr. Célio Cardoso Guimarães

À minha família, por todo amor e compreensão que sempre me deram.

Agradecimentos

A Deus, por sempre me dar forças e esperança.

Ao professor Célio, pela excelente orientação.

Aos meus amigos, pelo apoio e compreensão.

À Fundação de Amparo à Pesquisa do Estado de São Paulo, FAPESP (processo nº 96/09743-9), e ao Conselho Nacional de Pesquisa, CNPq, por financiarem parcialmente este trabalho.

Resumo

Log-structured File Systems, ou LFS, são sistemas de arquivos que possuem alto desempenho em ambientes de engenharia, escritórios e ensino, pois organizam os dados na forma de um *log* no disco. Seu alto potencial de aumento de desempenho provem da eliminação de grande parte dos posicionamentos do braço do disco (*seeks*) nas escritas e do uso tradicional de processadores e memória *cache* nas leituras.

Nesta dissertação, a estrutura do LFS é examinada em detalhes, analisando-se as suas duas principais implementações [ROS92a, SEL93a], e comparada com o sistema de arquivos tradicional do UNIX, o FFS [MCK84], no que diz respeito à organização dos dados, funcionamento e desempenho. As vantagens e desvantagens do LFS em relação ao FFS também são apresentadas. Diversos aspectos de sistemas de arquivos, como tecnologias de *hardware*, *cache* de dados e a camada VFS (*Virtual File System*) [KLE86] são analisados.

Neste trabalho, apresentamos a construção de um protótipo de um LFS no sistema operacional LINUX. Para este fim, utilizamos a ferramenta UserFS [FIT93], que possibilita a processos de usuário responderem às chamadas de sistema normalmente feitas apenas ao código do sistema de arquivos implementado no núcleo do sistema operacional.

Abstract

Log-structured File Systems (LFS) have high performance in workloads found in offices, engineering and educational environments, due to the physical data organization in a log, i.e. , a sequential area in the disk. It's high potential for performance improvements come from the elimination of disk seeks in write operations and the traditional use of processors and memory cache in read operations.

This dissertation examines in detail the structure of LFS and compares it to the traditional UNIX Fast File System (FFS) [MCK84], with respect to physical data organization internal workings and performance. The two main known LFS implementations [ROS92a, SEL93a] are also reviewed and compared. Many file system aspects such as hardware technologies, cache memories and the Virtual File System (VFS) [KLE86] are also examined.

In this work we built a LFS prototype for the LINUX operating system. For this purpose we have used the UserFS tool [FIT93]. This tool allows a user process to act on system calls usually handled by file system code inside the operating system kernel.

Conteúdo

Lista de figuras

Lista de tabelas

Capítulo 1

Introdução

Os sistemas de arquivos tradicionais, que normalmente são utilizados nos sistemas computacionais atuais, não foram projetados para as necessidades das novas aplicações que estão surgindo nos últimos anos. Tampouco estes sistemas são capazes de aproveitar integralmente as novas tecnologias desenvolvidas, tais como RAID (*Redundant Array of Inexpensive Disks*) e a melhora no desempenho dos componentes eletrônicos. Desta forma, novos modelos de sistemas de arquivos são exigidos atualmente.

Assim, aproveitando as tendências atuais de aumento de velocidade dos processadores e do tamanho da memória principal, foi desenvolvido um novo sistema de arquivos, chamado LFS (*Log-structured File System*) [ROS92b]. O LFS foi originalmente implementado para o sistema operacional Sprite e possui semântica igual à do sistema de arquivos do UNIX. A organização física dos dados, porém, é radicalmente inovadora, uma vez que a principal estrutura de armazenamento é um *log*.

Embora vários trabalhos tenham sido feitos sobre o LFS [BLA95, BUR92, SEL93a, SEL95], este ainda é um modelo controvertido, especialmente no que diz respeito ao seu desempenho e quanto à recuperação de falhas. Certos resultados obtidos mostram que o seu desempenho pode diminuir em até 45% em ambientes onde o sistema de arquivos não possui intervalos de tempo ociosos para que seja realizada a limpeza do *log* (seção 3.1.4). Esta dissertação traz uma descrição detalhada deste modelo e uma análise de seus pontos fortes e de seus pontos fracos. Seu desempenho também é analisado, em comparação ao sistema de arquivos FFS (*Fast File System*) [MCK84], em diferentes cargas de trabalho.

A principal motivação para estudarmos o LFS são os bons resultados de desempenho obtidos com este modelo quando utilizado em *workloads* encontrados em ambientes de engenharia, de ensino e escritórios, onde o sistema pode realizar a limpeza do *log* sem contenção com outras aplicações. Outra boa razão é o seu maior potencial para se beneficiar do aumento de desempenho dos componentes eletrônicos, tais como processadores e memória principal, se comparado a sistemas tradicionais [ROS92b].

Por diversas vezes nesta dissertação, o LFS é comparado com outros sistemas de arquivos, em especial com o FFS. A escolha do FFS como ponto de referência foi feita pois este é o sistema de arquivos mais utilizado atualmente nos ambientes UNIX. Suas

implementações são bastante testadas e existem diversos trabalhos analisando o seu desempenho em diversos ambientes diferentes [MCV91, SEL95].

Para contribuir com o desenvolvimento do LFS e comprovar as qualidades deste tipo de sistema de arquivos, construímos uma versão sua no sistema operacional LINUX. Implementamos este sistema de arquivos como um protótipo fora do núcleo do sistema operacional, devido à complexidade de implementá-lo como um sistema de arquivos comum. O protótipo é representado por um processo de usuário que se comunica com o núcleo do sistema operacional através de uma ferramenta chamada UserFS [FIT93]. Embora o protótipo não corresponda a um sistema de arquivos completo e estável, este possui todas as principais estruturas do LFS e implementa a maior parte das operações exigidas pelo VFS (*Virtual File System*) do LINUX.

1.1 Organização da dissertação

O restante desta dissertação está organizado da seguinte forma. O Capítulo 2 é uma revisão das principais tecnologias empregadas no nosso trabalho. São tratados neste capítulo o sistema de arquivos FFS e o VFS, mecanismo que permite o suporte a diversos sistema de arquivos diferentes por um mesmo núcleo de sistema operacional.

O texto concentra-se nas soluções de *software* encontradas na literatura, pois é nesta área que o trabalho está inserido, mas são tratados também alguns aspectos da organização física dos discos que influenciam no projeto e no desempenho dos sistemas de arquivos. Algumas tecnologias de *hardware* que vão de encontro ao modelo do LFS, como é o caso dos dispositivos RAID e SSD (*Solid State Disk*) [CAS98], são explanadas. São abordadas também as necessidades existentes hoje e as que deverão surgir na área de sistemas de arquivos, bem como de que forma estas necessidades influenciam no seu desenvolvimento.

Este capítulo não pretende ser uma referência completa sobre as tecnologias empregadas atualmente em sistemas de arquivos. Diversos aspectos, como sistemas de arquivos distribuídos e sistemas de arquivos em ambientes operacionais diferentes do UNIX não são abordados aqui. Os objetivos do capítulo são fornecer a motivação para estudarmos o LFS e informações para o melhor entendimento dos próximos capítulos, que descrevem com maior profundidade o LFS e o sistema experimental que desenvolvemos.

O Capítulo 3 explica o LFS. São tratadas em detalhes a organização física dos dados e metadados no disco bem como as estruturas que compõem este sistema de arquivos. O sistema também é comparado ao FFS em termos de organização em disco, dos algoritmos que determinam seu comportamento e de seu desempenho em diversos *workloads*.

As diferenças mais relevantes existentes entre as duas principais implementações do LFS [ROS92a, SEL93a] são ressaltadas. São explicados também alguns trabalhos mais recentes feitos sobre o LFS que agregam novas funcionalidades e comprovam a flexibilidade do modelo.

No Capítulo 4 é descrito o ambiente alvo da implementação do protótipo, que é o sistema operacional LINUX. O texto aborda em maior profundidade as características do

sistema de arquivos nativo do LINUX, o Ext2, e outros aspectos que influenciaram o trabalho, tais como sua *buffer cache* e a camada VFS.

É explicada também no Capítulo 4 a ferramenta UserFS, que foi vital para a implementação do protótipo. Esta ferramenta favorece a prototipação de sistemas de arquivos fornecendo uma ponte entre um processo de usuário que implementa o sistema de arquivos e o núcleo do sistema operacional.

O Capítulo 5 trata do desenvolvimento do trabalho em si. Este inicia-se com a explicação das estruturas que compõem o sistema de arquivos desenvolvido, chamado ULFS, detalhando a composição de cada uma destas estruturas e mostrando os relacionamentos que possuem entre si. É explicado também o funcionamento do protótipo que implementa o sistema de arquivos em situações como as chamadas de sistema `mount` e `umount`, a limpeza de segmentos e a recuperação de falhas. O Capítulo traz ainda os testes de correção e desempenho realizados sobre o protótipo. Por último, é feita uma análise do projeto e são propostas algumas extensões como trabalhos futuros.

Em vários locais no texto, utilizamos formatações diferentes para destacar algumas palavras e expressões. O padrão que utilizamos é o seguinte: palavras e expressões em Inglês são grafadas em itálico, como em *workload*. Nomes de comandos do UNIX, constantes padrão do sistema, tipos de dados em C e nomes de programas são grafados com caracteres monoespaço, como em `mount`. Nos títulos das seções e nas legendas de figuras e tabelas, todo o texto segue a formatação padrão.

Capítulo 2

Sistemas de arquivos modernos

Mesmo com o desenvolvimento de tecnologias de armazenamento de dados baseadas em dispositivos semicondutores, como memórias RAM não voláteis, os discos magnéticos continuam sendo, sem sombra de dúvida, o mais importante meio de armazenamento permanente de grandes volumes de dados para sistemas computacionais. Além da característica de não volatilidade, uma das principais vantagens dos discos magnéticos é seu preço reduzido. Isto permite que mesmo equipamentos pessoais baratos possuam hoje alguns gigabytes de armazenamento secundário. Os discos magnéticos também são relativamente confiáveis. Atualmente, podem ser encontrados discos comerciais nos quais o tempo médio para falha, ou MTTF (*Mean Time To Failure*), encontra-se próximo de 10^6 horas [SEA98, QUA98a]. Maior confiabilidade também pode ser acrescentada utilizando-se técnicas de replicação [CHE94].

No entanto, o desenvolvimento acelerado na tecnologia dos circuitos digitais, especialmente processadores e memórias, tem causado grande pressão para o aprimoramento dos mecanismos de armazenamento de massa. Com processadores cada vez mais rápidos e memórias RAM cada vez maiores e mais rápidas, os sistemas computacionais tendem a ficar mais e mais limitados pela velocidade dos discos magnéticos [OUS90].

Embora o desenvolvimento do *hardware* na área de armazenamento secundário também seja expressivo, o ganho em velocidade dos discos nos últimos anos não tem acompanhado o dos circuitos digitais. Por esta razão, o projeto de sistemas de arquivos mais rápidos e adaptados tanto às características atuais de *hardware* quanto às exigências das aplicações atuais, e das que estão surgindo, é de suma importância para o aumento no desempenho dos sistemas computacionais modernos.

Neste capítulo, serão descritos os desafios existentes atualmente para a produção de novos sistemas de arquivos, bem como algumas tecnologias utilizadas para aumentar o seu desempenho ou para tornar o subsistema de arquivos mais flexível, como é o caso do VFS. O capítulo termina com uma seção sobre a evolução do sistema de arquivos do UNIX. Serão apresentadas as características mais importantes de três sistemas de arquivos que melhor representam os estágios de desenvolvimento do FFS, o modelo mais utilizado atualmente entre os sistemas operacionais UNIX.

2.1 Necessidade de novos sistemas de arquivos

A maior motivação para o desenvolvimento de novos sistemas de arquivos é a discrepância entre a velocidade com que os processadores atuais podem resolver as requisições de entrada e saída das aplicações e a velocidade, bem menor, com que os discos magnéticos podem responder a estas requisições.

Embora o avanço na tecnologia dos discos magnéticos nas últimas décadas tenha sido relevante, este não tem acompanhado o crescimento no desempenho dos processadores e da memória principal. Enquanto a velocidade dos processadores em milhões de instruções por segundo (MIPS) cresce à uma taxa de aproximadamente 50% ao ano [HEN94], o desenvolvimento dos discos magnéticos concentra-se no aumento da densidade dos dados e na diminuição do custo por byte armazenado. Isto tende a deixar os sistemas computacionais atuais limitados pelas operações de disco [ROS92a, CHE94].

Além do problema na diferença entre o desempenho dos discos magnéticos e dos processadores, outro incentivo para o desenvolvimento de sistemas de arquivos é o aparecimento de novas aplicações. Com a disseminação da Internet, a utilização de multimídia e a integração entre os meios de comunicação e a computação, novos usos para os computadores surgem a cada dia. Esses incluem servidores de áudio e vídeo digitais para companhias de distribuição de canais por assinatura, aplicações com utilização maciça de arquivos gráficos e de áudio e servidores de páginas HTML. Essas tecnologias desenvolvidas recentemente exigem o tratamento de um montante elevado de dados, muitas vezes em tempo real, o que determina um *throughput* que os sistemas de arquivos convencionais não suportam.

É muito difícil, se não impossível, para um sistema de arquivos possuir todas as características exigidas por qualquer tipo de aplicação. O desenvolvimento de um bom sistema de arquivos demanda compromissos. Por exemplo, o uso de replicação de dados e mecanismos de *checksum,* com o intuito de aumentar a confiabilidade e integridade dos dados armazenados, pode diminuir bastante o *throughput* geral do sistema. Da mesma forma, utilizar *cache* de dados e metadados em memória RAM para melhorar o tempo de resposta pode deixar o sistema de arquivos mais vulnerável a perdas e inconsistências dos dados, devido a falhas de equipamento e quedas de alimentação.

Todo projeto de sistema de arquivos deve levar em conta as condições nas quais este será utilizado e os requisitos que se deseja deste sistema. Por exemplo, aplicações científicas que manipulam grandes quantidades de dados estão muito mais preocupadas com o *throughput* do que com a confiabilidade de armazenamento. Uma simulação pode ser repetida, caso ocorra uma queda do sistema durante a sua realização. Por outro lado, um ambiente de engenharia de *software* com muitos usuários, uso de transações longas, controle de versões e trabalho cooperativo, necessita de mecanismos de suporte a transações, confiabilidade dos dados e volume de operações de E/S. Caso o sistema de arquivos não apresente estas qualidades, este deve ao menos prover meios para que sejam implementadas em um nível mais alto, sem que a perda de desempenho gerada torne seu uso inviável.

2.2 Workloads

Um fator muito importante no projeto de sistemas de arquivos é a carga de trabalho, ou *workload*, à qual o sistema será submetido. O *workload* envolve tanto características dos dados, tais como o tamanho médio dos arquivos e o número de arquivos no sistema, como características da utilização destes dados, tais como a taxa média de operações de E/S por período de tempo e o padrão de acesso aos dados (que pode ser seqüencial ou aleatório), entre outras.

Quanto às características dos dados, pode-se tomar, por exemplo, o tamanho dos arquivos para enfatizar o impacto do *workload* no projeto de sistemas de arquivos. Arquivos grandes são mais difíceis de serem mantidos fisicamente contíguos, pois necessitam de grandes regiões livres de disco. Sucessivas operações de criação e remoção de arquivos tendem a acabar com estas regiões. Por outro lado, arquivos muito pequenos podem aumentar a fragmentação interna dos blocos de dados, ou seja, o espaço perdido no último bloco de cada arquivo. Além disso, como haverá mais arquivos no mesmo espaço da unidade de armazenamento, mais metadados serão necessários para mantê-los, diminuindo o espaço livre para dados do usuário. Sistemas de arquivos otimizados para arquivos grandes, como é o caso do FFS com utilização de agrupamentos (*clusters*), sofrem uma queda substancial no desempenho quando manipulam arquivos que ocupam até um bloco e tendem a deixar o espaço livre em disco demasiadamente fragmentado [SEL95]. Sistemas de arquivos otimizados para arquivos pequenos, como é o caso do LFS, podem ter uma queda de desempenho elevada na manipulação de arquivos grandes.

O padrão de acesso aos arquivos também é de suma importância para o desempenho de um sistema. Os arquivos muitas vezes não são criados e salvos de uma só vez no disco. Ao invés disso, as aplicações geralmente gravam pequenas partes dos arquivos, normalmente algumas centenas de bytes. Assim, fica difícil para o sistema de arquivos alocar blocos de dados muito grandes para os arquivos, já que a gravação de apenas um byte novo no arquivo determina a gravação de um bloco de dados inteiro no disco. Além disso, arquivos gravados de forma aleatória são mais difíceis de serem alocados seqüencialmente, pois o alocador do sistema de arquivos não sabe quanto o arquivo vai crescer e fica difícil decidir quanto espaço reservar para o arquivo. Alguns sistemas de arquivos baseados em extensões – *extent based file systems* [VAH96] – ou seja, que alocam arquivos em grandes regiões e não em blocos, utilizam pistas fornecidas pelas aplicações para saber qual o tamanho médio que os arquivos alcançarão, de forma a reservar espaço suficiente para estes. Ainda, se arquivos alocados aleatoriamente forem lidos de forma seqüencial, o desempenho do sistema pode ser bastante degradado (seção 2.4.2).

O *workload* de um sistema pode mudar com o tempo. Por exemplo, uma aplicação pode criar um arquivo e gravar seus blocos seqüencialmente, até atingir um tamanho de alguns gigabytes. Esta pode fazer isso com diversos arquivos e então terminar. Outra aplicação pode então ler os blocos destes arquivos de forma aleatória enquanto cria centenas de arquivos pequenos com os dados lidos. Mas, em geral, um ambiente computacional possui

um *workload* característico, onde a grande maioria dos acessos a arquivos seguem um padrão bem definido.

Um dos *workloads* mais difíceis para os sistemas de arquivos é o encontrado em ambientes de engenharia e em escritórios. Estudos mostram que neste tipo de ambiente a maioria dos arquivos são pequenos, com apenas alguns kilobytes [OUS85]. Outra característica é que os arquivos tendem a ser lidos inteiramente, modificados e gravados em disco também de forma integral. Apesar deste padrão seqüencial de acesso, o tamanho reduzido dos arquivos torna a maioria das operações de leitura e escrita onerosas para o sistema de arquivos, pois é necessário ler e atualizar os metadados associados a cada arquivo manipulado. Dependendo da organização física dos dados e metadados, a queda de desempenho pode ser tão grande que apenas 5% da faixa de passagem (*bandwidth*) total do disco pode ser efetivamente utilizada para dados (seção 2.4.1). Outra característica importante deste tipo de ambiente é que cerca de 25% das novas informações criadas são removidas em no máximo 30 segundos e 50% é removida em no máximo 5 minutos [OUS85]. Assim, o sistema pode gastar um bom tempo realizando operações desnecessárias. Geralmente, os sistemas de arquivos mais novos aguardam 30s depois que um bloco de dados é atualizado para gravá-lo no disco.

Normalmente, nos ambientes de engenharia e escritório, os arquivos não são referenciados com a mesma freqüência. Ao contrário disso, alguns poucos arquivos, geralmente 10% do total, recebem cerca de 90% das operações realizadas, enquanto os outros 90% recebem os 10% restantes. Os arquivos que sofrem mais acessos representam aplicações do sistema, como comandos implementados fora do interpretador, ferramentas rotineiras (como correio eletrônico, compiladores e processadores de texto) e dados dos usuários. Isso, por um lado significa que mesmo *caches* pequenas em memória principal podem manter a maior parte dos dados requisitados pelas aplicações. Mas por outro lado, este padrão de acesso pode transformar estes arquivos mais requisitados em um gargalo para o sistema.

2.3 Tecnologias empregadas

Como foi visto acima, a pressão sobre o desempenho dos sistemas de arquivos é muito grande, uma vez que os sistemas atuais tendem a ficar limitados pela velocidade de resposta dos discos. A seguir, será apresentada uma breve explanação sobre as características físicas dos discos magnéticos e como estas influenciam no desempenho dos sistemas de arquivos. Também serão apresentadas algumas tecnologias novas e outras já bem solidificadas, tanto de *hardware* como de *software*, que visam melhorar o desempenho dos sistemas de armazenamento secundário em geral. Estas tecnologias foram escolhidas por serem especialmente importantes para o modelo do LFS e para a implementação do nosso protótipo, mas a maioria serve a qualquer modelo de sistema de arquivos.

2.3.1 Discos magnéticos

Os discos magnéticos são inerentemente mais lentos do que os circuitos eletrônicos, pois são dispositivos eletromecânicos que possuem partes móveis que dependem de limitações tecnológicas e físicas. De modo geral, como é mostrado na Figura 2.1, uma unidade magnética de armazenamento é composta de uma série de **discos** (ou pratos) empilhados e conectados a um eixo comum. Uma vez iniciada a unidade, o eixo, e em conseqüência os discos, gira com velocidade e sentido constantes, enquanto for fornecida alimentação. O diâmetro dos discos atualmente é de 3½ ou 5¼ polegadas e a velocidade de rotação varia entre 5400 e 11000 RPM. No início dos anos 80, a velocidade de rotação mais comum era de 3600 RPM.

A alguns milésimos de milímetro da superfície de cada prato, encontra-se uma **cabeça de leitura/gravação**, capaz de detectar ou proporcionar mudanças na polaridade de grupos de moléculas do material magnético que recobre a superfície dos pratos. Estas mudanças representam as informações contidas no disco e são convertidas em bytes por circuitos análogo-digitais. Todas as cabeças estão ligadas a um único acionador e movimentam-se radialmente em conjunto.

A superfície do disco é dividida em **trilhas** concêntricas que correspondem às posições onde a cabeça de leitura/gravação pode se posicionar. A quantidade de trilhas depende da densidade de gravação e do diâmetro do disco. O conjunto das trilhas de mesma posição em cada disco é denominado **cilindro**. Cada trilha é dividida em **setores**, geralmente com 512 bytes cada. Um setor é a menor unidade de alocação de um disco, mas os sistemas operacionais atuais geralmente não manipulam apenas um setor por operação de entrada e saída. O que estes fazem é ler ou escrever conjuntos de setores adjacentes chamados **blocos lógicos** dos sistemas de arquivos. A quantidade de setores por trilha também depende da densidade e do diâmetro dos discos. Alguns discos conseguem alocar mais setores nas trilhas mais externas, pois estas possuem um perímetro maior que as mais próximas do centro. Os controladores escondem esta condição do sistema operacional, para o qual o disco possui sempre um número igual de setores por trilha. Assim, para o sistema operacional, os endereços dos setores são físicos, mas para os controladores estes são lógicos e devem ser traduzidos nos endereços reais em disco.

Existem diversos fatores que influenciam o desempenho de uma unidade de armazenamento secundário e existem diversas formas de se medir este desempenho. Uma das principais medidas que identificam a capacidade de um disco é a taxa de transferência de dados, ou seja, a quantidade de bytes transferidos entre o disco e o barramento, dividida pela unidade de tempo. A taxa de transferência depende diretamente da densidade da trilha e da velocidade de rotação do disco e indica a quantidade máxima de dados que o disco consegue transferir após estar devidamente posicionado. Esta não leva em conta o tempo gasto para posicionar a cabeça numa certa trilha. Dependendo do modelo do disco, a taxa de transferência atualmente varia de 4 a 20MB/s.

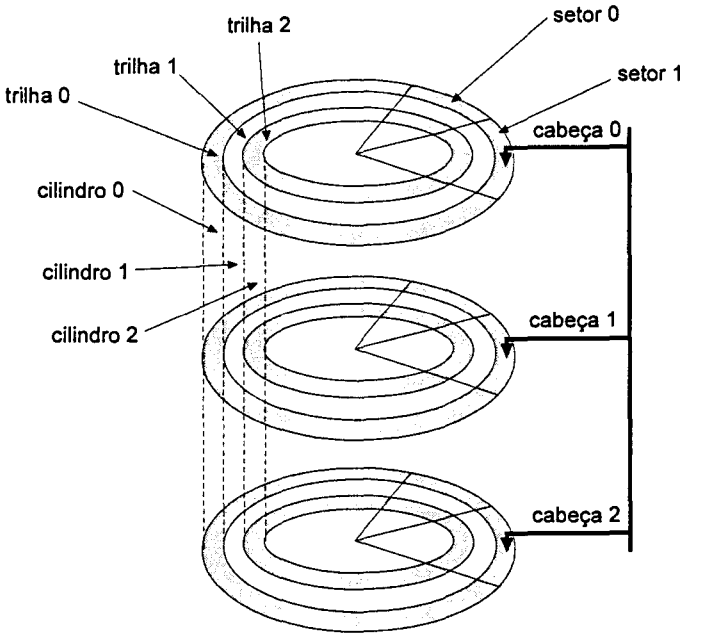


Figura 2.1: Visão esquemática de um disco magnético.

Para ter acesso a um setor específico do disco, as cabeças de leitura e gravação devem ser posicionadas no cilindro onde se encontra o setor desejado. O tempo gasto para isso é chamado tempo de posicionamento (*seek time*). Se o bloco desejado estiver no mesmo cilindro onde as cabeças do disco se encontram, o tempo de posicionamento será zero. Caso contrário, o melhor caso é quando o bloco desejado estiver em um cilindro adjacente ao qual as cabeças do disco estão. Neste caso, o tempo gasto recebe o nome de posicionamento trilha à trilha. O tempo de posicionamento médio de um disco é obtido pela média dos tempos de posicionamento em um acesso aleatório a setores espalhados pelo disco e atualmente gira em torno de 7 a 10ms.

Posicionar a cabeça de leitura/gravação não é o único atraso sofrido pelos discos. Uma vez na trilha certa, a unidade precisa esperar que o setor desejado esteja abaixo da cabeça de leitura/gravação para que possa ser lido ou escrito. O tempo de espera é chamado latência rotacional (*rotational latency*) e seu valor é em média meio período de rotação do disco.

Em certos casos, o sistema operacional lê apenas um bloco por vez, o que pode ocorrer devido às limitações do sistema operacional, por limitações dos controladores e barramentos ou pelo padrão de acesso da aplicação que está solicitando o bloco. Se isso ocorrer, mesmo que os próximos blocos solicitados estejam na mesma trilha e sejam posteriores ao primeiro bloco lido, o sistema será obrigado a esperar em média meia revolução para poder ler cada um dos próximos blocos. Para evitar este atraso a cada bloco, os discos mais novos possuem um *buffer* de trilha. Quando as cabeças são posicionadas em um cilindro e um setor é lido, todos os demais são lidos e colocados no *buffer* de trilha. Assim, quando o próximo bloco for requisitado, este já terá sido lido.

A latência rotacional e o tempo de posicionamento prejudicam muito o desempenho dos discos magnéticos. Dependendo de quão aleatório é o acesso aos blocos, a taxa de transferência real pode ser bem menor do que a taxa de transferência anunciada. Assim, os sistemas de arquivos devem dispor os blocos dos arquivos em disco de forma a minimizar o impacto destes atrasos. A alocação de blocos é o próxima técnica apresentada.

2.3.2 Alocação inteligente de arquivos

Como já foi visto, os discos magnéticos são mais lentos que os dispositivos eletrônicos. Assim, se as aplicações realizam muitas operações com arquivos, o processador pode ficar um bom tempo ocioso, aguardando que estas sejam completadas pelo disco. Desta forma, os novos sistemas de arquivos devem ser projetados balanceando a utilização do disco e do processador, de acordo com a velocidade de resposta e disponibilidade de cada um. Ou seja, os novos sistemas de arquivos devem utilizar-se mais dos processadores, normalmente subutilizados, e tornar os acessos a disco menos rotineiros e otimizados. Isso é feito através de algoritmos inteligentes para a alocação dos blocos de dados dos arquivos, de forma que o tempo gasto pelo processador para encontrar a posição de gravação mais adequada seja compensado por uma menor movimentação da cabeça do disco e um atraso rotacional menor.

Na memória RAM o acesso a qualquer palavra demora sempre o mesmo intervalo de tempo, não dependendo do acesso anteriormente efetuado. Ao contrário disso, o tempo gasto nos discos para posicionar a cabeça de leitura/gravação na trilha certa e esperar o disco girar até que o setor desejado fique sob a cabeça depende muito do último acesso. Somado a isso, os usuários tendem a realizar operações simultaneamente em vários arquivos que possuem algum relacionamento entre si. Por exemplo, toda vez que um projeto é compilado e ligado, os mesmos arquivos fonte e bibliotecas pré-compiladas são sempre acessados. Geralmente os usuários guardam estes arquivos em árvores de diretórios, formando uma organização lógica que facilita a sua localização. Caso estes arquivos estejam espalhados pelo disco de forma aleatória, o tempo gasto fazendo movimentações da cabeça de leitura/gravação será muito maior que o tempo gasto transferindo bytes para (ou a partir do) sistema operacional. Desta forma, um bom sistema de arquivos deve aproveitar-se ao máximo da localidade de referência das aplicações, alocando, por exemplo, arquivos que pertencem a um mesmo diretório em uma mesma região do disco.

Outro padrão de acesso comum em diversos sistemas, que vão desde computação científica até ambientes de engenharia e de escritórios, é a transferência de arquivos inteiros, sejam estes grandes ou pequenos. Assim, os blocos de um mesmo arquivo devem estar sempre que possível próximos uns dos outros. Também é essencial que a alocação dos blocos respeite a posição lógica dentro do arquivo, aproveitando a rotação do disco.

O sistema de arquivos FFS (seção 2.4.2) utiliza o conceito de grupos de cilindros para agrupar arquivos relacionados e tenta gravar sempre que possível os blocos dos arquivos seqüencialmente no disco.

2.3.3 Memória cache e VMS

A maior parte dos acessos a sistemas de arquivos são requisições de leitura [OUS85]. Assim, os sistema operacionais contam com *caches* de memória suficientes para manter os arquivos mais requisitados do sistema de arquivos em memória. Isso diminui substancialmente a quantidade de acessos a disco, uma vez que apenas os dados não encontrados nas caches devem ser procurados no disco.

O uso de *cache* implica na necessidade de se aumentar a quantidade de memória dos computadores, uma vez que a memória que estaria disponível para as aplicações fica reservada pelo sistema operacional para guardar cópias das informações em disco. Por sorte, as memórias RAM aumentam a sua capacidade e diminuem seu preço numa taxa maior do que os discos magnéticos. Isto permite que os novos sistemas possuam cada vez mais memória principal.

Uma *cache* de sistema de arquivos normalmente é dividida em *buffers*, daí o nome ***buffer cache***, dado a este tipo de *cache*. Os *buffers* podem possuir diversos tamanhos dentro da *cache*, mas os *buffers* de um sistema de arquivos possuem sempre o mesmo tamanho, geralmente apenas um bloco de dados. Cada *buffer* possui um cabeçalho (*header*) com informações sobre sincronização, tamanho do *buffer* e *flags* necessários aos algoritmos de gravação e recuperação dos blocos. Muitas vezes, o cabeçalho ainda guarda o arquivo ao qual pertence o bloco e a posição lógica do bloco dentro do arquivo. Assim, o sistema pode encontrar facilmente os blocos de um arquivo na *cache*. Alguns sistemas podem indexar os blocos de forma diferente (seção 4.1.2).

Toda requisição de E/S passa primeiro pela *cache*. O deslocamento dentro do arquivo e a quantidade de bytes a serem lidos ou escritos são traduzidos para os blocos lógicos que devem ser procurados. Caso um ou mais blocos procurados não sejam encontrados, as rotinas de tratamento de ausência de blocos alocam novos *buffers* na *cache* e lêem os blocos do disco. Quando a *cache* se enche e o sistema necessita ler mais blocos, os *buffers* que foram requisitados há mais tempo são requisitados para serem ocupados por novos dados. Caso as informações que os *buffers* contenham tenham sido modificadas, estes devem ser salvos no disco. Senão, o *buffer* pode ser utilizado imediatamente para os novos blocos de dados.

Mesmo tendo seu custo diminuído, a memória principal continua sendo um recurso valioso e escasso nos sistemas computacionais. Se a *cache* utilizar memória demais, as aplicações de usuário e até mesmo outros subsistemas do núcleo do sistema operacional não terão espaço para alocar suas estruturas. Por outro lado, uma *cache* muito pequena não pode armazenar muitos dados e a taxa de ausência (*miss rate*) será muito grande, diminuindo a sua eficiência. Desta forma, os sistemas operacionais modernos, como o Mach, BSD 4.4 (*Berkeley Software Distribution 4.4*) e SVR4 (*System V Release 4*), integram as *caches* do sistema de arquivos ao sistema de memória virtual, ou VMS (*Virtual Memory System*) [VAH96]. Com o VMS, o sistema operacional possui uma visão uniforme da memória principal, pois os dados provenientes do disco são alocados em páginas, da mesma forma que as páginas requisitadas por aplicações e pelo núcleo. Assim, o sistema não separa uma parte específica da memória para a *cache* de disco e o espaço é alocado para o subsistema de arquivos conforme

necessário. Da mesma forma, quando o núcleo necessita de mais espaço de memória, o subsistema de memória reutiliza as páginas para outros fins, como dados dos processos. No caso das páginas estarem sujas, ou seja, conterem dados dos arquivos que foram modificados, o seu conteúdo é salvo em disco antes destas serem liberadas.

Além da memória RAM e do sistema de arquivos, o VMS cuida também das áreas de *swap* do sistema, que são regiões das unidades de armazenamento secundárias onde o sistema operacional pode salvar páginas que estão em memória. O VMS utiliza áreas de *swap* para aumentar a memória virtual do sistema. Quando acaba o espaço em memória RAM, o VMS grava o conteúdo de algumas páginas no *swap* e as utiliza para novos dados. Se alguma aplicação requisitar uma página que foi transferida para o *swap* (*swapped out*), o sistema transfere outra página da memória para o *swap* e copia a primeira página novamente na memória. O VMS mantém tabelas em memória para saber quais páginas estão em memória, quais estão no *swap* e quais foram modificadas, dentre outras informações que necessita. A implementação destas tabelas varia muito entre os sistemas operacionais e depende também dos suportes de *hardware*, tais como *chips* MMU (*Memory Management Unit*), que as plataformas provêm.

A integração entre o sistema de arquivos e o sistema de memória virtual também permite que partes de arquivos sejam mapeados diretamente no espaço de endereçamento de um processo, através de uma chamada de sistema `mmap`, ou seja, *memory map*. Assim, a aplicação economiza diversas chamadas de sistema e salva espaço em memória principal. Por exemplo, em sistemas que não possuem mapeamento de arquivos, quando um processo lê um arquivo, seus blocos são primeiramente colocados na *cache* e, para cada porção do arquivo requisitado pela aplicação, o sistema faz uma cópia entre a *cache* e uma página de dados no espaço do processo. Caso o processo faça alguma modificação nos dados e queira gravá-la no arquivo, o sistema deve copiar os dados novamente para a *cache* e então para o disco. Em sistemas com mapeamento de arquivos, o processo pede ao sistema para mapear uma porção do arquivo, que é dividida em páginas do VMS e colocada no espaço do processo. Quando ele quer modificar algum dado, este o faz utilizando instruções comuns de memória, sem necessidade de chamar `write` no final da operação. O VMS identifica as páginas que estão sujas, realizando as alterações necessárias nos metadados, isto é, no inode do arquivo, e liberando as páginas quando lhe convier. Para realizar o mapeamento de arquivos, o VMS precisa estar totalmente integrado ao subsistema de arquivos.

Se mais de um processo requisitar o mapeamento do mesmo arquivo em memória, as regiões mapeadas coincidentes são representadas por páginas compartilhadas. Qualquer modificação que um processo faz nestas páginas é imediatamente vista pelos outros processos que as compartilham. Isso difere da semântica das operações normais de I/O, as quais bloqueiam temporariamente os blocos e metadados que estão alterando, até que todos os bytes sejam gravados. Algumas implementações fornecem um mapeamento exclusivo de uma região para um processo. Desta forma, o processo recebe o mapeamento em uma página que só pode ser alterada por ele mesmo. Mas isso não impede que outros processos realizem mapeamentos, tanto compartilhados como exclusivos, sobre a mesma região do mesmo arquivo. Quando um mapeamento exclusivo é desfeito, o arquivo não é atualizado e os dados alterados na página se perdem.

Muitos sistemas dependem da gravação imediata dos dados de um arquivo para manter a consistência do sistema de arquivos. Assim, a *cache* deve estar preparada para gravar imediatamente qualquer bloco de dados que for modificado, mantendo a cópia atualizada dos dados tanto em memória quanto no disco. Estas *caches* recebem o nome de *write-through*. O problema de *caches write-through* é que a operação *write* é síncrona, ou seja, a aplicação fica esperando o sistema completar a operação. Para solucionar este problema, muitos sistemas utilizam *caches write-behind*, as quais armazenam a informação em memória e gravam-na assincronamente quando, por exemplo, não houver contenção pelo disco. Se ocorrer alguma falha do sistema, como uma queda de energia, enquanto os dados não estiverem salvos, estes serão perdidos. Porém, os metadados de um sistema de arquivos são muito críticos para seguir esta política. Dependendo dos metadados contidos na *cache* no momento da falha, o sistema de arquivos pode ficar bastante danificado e até mesmo irrecuperável. Assim, sistemas como o UFS (seção 2.4.3) gravam os metadados imediatamente no disco. Outros sistemas utilizam um *log* temporário para guardar informações sobre as operações de escrita [SEL92, VAH96]. Se ocorrer uma falha antes dos metadados serem salvos, as operações podem ser reconstituídas através do *log*.

Outra forma para impedir que as *caches* percam informações devido a quedas de energia é faze-las não voláteis. Várias técnicas podem ser utilizadas para implementar tais *caches*. Uma delas é usar memórias RAM não voláteis (NV-RAM), o que aumenta consideravelmente o preço do sistema. Outra solução é acoplar uma bateria em uma unidade de memória RAM, de forma que os dados fiquem protegidos até que a energia seja restabelecida. O perigo desta técnica é que as baterias podem se esgotar antes que o problema da energia seja solucionado. Por último, pode-se utilizar uma unidade de memória RAM com um disco magnético acoplado. Estas unidades, chamadas *solid state*, possuem baterias suficientes para transferir os dados existentes na *cache* para o disco em caso de falta de energia. Após a energia ser restabelecida, o disco é lido e os dados são transferidos novamente para a memória RAM, restabelecendo o estado anterior do sistema [QUA98b, CAS98]. As unidades que compõem esse tipo de memória não volátil são acopladas ao sistema como se fossem um disco comum e são fornecidas atualmente com capacidades que vão de 130MB a 2.5GB. Em aplicações críticas, onde a segurança dos dados compensa o preço pago na tecnologia, todo o armazenamento pode ser conseguido através de unidades *solid state*. Porém, como seu custo ainda é muitas vezes o preço de uma unidade comum, o mais indicado é utilizar estas unidades para armazenar apenas os dados mais acessados do sistema. Por exemplo, uma unidade destas poderia corresponder à área de *swap*, armazenando páginas do sistema de memória virtual.

Embora as caches não voláteis sejam eficientes na segurança dos dados, sua utilização não é tão simples. Os sistemas operacionais precisam estar preparados para considerar os dados das *caches* não voláteis quando a energia do sistema é restabelecida e não reiniciá-las. Além disso, o uso de cache de tamanhos fixos pode atrapalhar sua integração com o sistema de memória virtual.

As *caches* dos sistemas de arquivos podem diferir também na natureza dos dados armazenados. As mais comuns são as *caches* de blocos, sejam de arquivos ou de metadados, como inodes, superblocos e mapas de bits. Um outro tipo de *cache* utilizado é a de diretórios.

Estas são usadas para facilitar o trabalho de mapear nomes de arquivos em vnodes, que são as estruturas que representam os arquivos abertos em memória (seção 2.3.5). As *caches* de diretório normalmente são formadas por entradas que contêm um apontador para o vnode do diretório, o nome do arquivo naquele diretório e um apontador para o vnode do arquivo. A *cache* é ordenada pelos diretórios e pelo nome dos arquivos. Assim, o sistema não necessita percorrer as entradas dos arquivos diretório até encontrar o arquivo que estava procurando. Os sistemas de arquivos do BSD 4.3 e do SVR4 utilizam *caches* de diretório [VAH96].

2.3.4 Redundant Arrays of Inexpensive Disks

Os RAIDs, ou *Redundant Arrays of Inexpensive*[1] *Disks*, são dispositivos de *hardware* desenvolvidos para aumentar o desempenho das unidades de armazenamento secundárias [CHE94]. Um RAID consiste de uma série de discos comuns colocados em paralelo e conectados a um único controlador que possui apenas um canal para transferência de dados. O principal objetivo das unidades RAID é aproveitar o paralelismo dos discos para aumentar o *throughput* da unidade. Se diferentes operações de E/S requerem dados que estão em discos distintos, estas podem ser servidas ao mesmo tempo, bastando que o controlador da unidade sincronize os comandos e a transferência dos dados.

Se cada um dos discos pode ser lido ou escrito em paralelo, uma forma natural de aumentar o *throughput* de um RAID é aumentar o número de discos existentes na unidade. Porém, aumentar o número de disco significa aumentar o risco de falhas de um disco em particular. Um RAID com 100 discos comuns está 100 vezes mais sujeito a falhas do que um único disco do mesmo modelo. A forma encontrada para aumentar a segurança dos RAIDs é replicar a informação contida neles, de forma que esta possa ser reconstituída através da informação redundante.

O método mais simples de replicação é fazer uma cópia de dada bloco de dados de um disco. Assim, cada disco do *array* possui um disco espelho (*mirror*) exatamente igual a si. Com esta solução, a capacidade do *array* fica reduzida à metade do número total de discos, o que a torna bastante cara. Além disso, o *throughput* total da unidade passa a ser a metade da soma dos *throughputs* individuais, uma vez que os espelhos só são usados para leitura quando ocorrer falha na sua "imagem real". Sistemas RAID que utilizam esta técnica de replicação são chamados de nível 1.

Uma outra opção é utilizar um dos discos para guardar somente a paridade dos demais discos do *array*. O cálculo de paridade pode ser realizado em bits (RAID nível 3), ou em blocos, chamados de *stripe unit* (RAID nível 4). Chamaremos de **linha** (*stripe*) o conjunto de bits ou blocos que possuem a mesma posição lógica em cada um dos discos do *array*. Nos RAIDs nível 3, todas as gravações fazem acesso a todos os discos de dados e o disco de paridade e as leituras fazem aceso a todos os discos de dados. Na ocorrência de uma falha em

[1] Na literatura, a letra "I" pode ser referenciada como *Industrial* ou *Independent*. Adotamos aqui o termo mais tradicionalmente utilizado.

uma das unidades, os bits contidos nas demais são suficientes para recompor a informação. Nos RAIDs nível 4, a paridade é calculada fazendo-se um "ou exclusivo" de cada bloco. Apenas acessos grandes são capazes de ler ou escrever em todos os discos do *array*. Quando apenas alguns blocos são gravados, a paridade de toda a linha é recalculada, levando-se em conta a informação que já existia nos blocos que estão sendo gravados. Desta forma, a gravação de um único bloco necessita de quatro acessos ao *array*: um para ler as informações antigas, um para ler a paridade da linha, um para gravar o bloco e um para gravar a nova paridade. Este procedimento recebe o nome de ler-modificar-gravar (*read-modify-write*) e é o maior problema com RAIDs com paridade de dados.

O uso de um disco dedicado a guardar apenas a paridade nos RAIDs nível 4 pode transformar este disco em um gargalo para o sistema, uma vez que o disco participa de qualquer escrita, mesmo aquelas que não utilizam todos os outros discos da linha. Além disso, em condições normais de operação, as leituras não aproveitam o disco de paridade. Uma solução é distribuir a paridade do *array* por todos os discos. RAIDs que fazem isso são chamados de nível 5 e são os mais utilizados atualmente para aplicações em ambientes de engenharia e escritórios, pois possuem boa relação entre custo e desempenho. Existem vários esquemas para distribuir a paridade em RAIDs nível 5, mas o mais utilizado chama-se *left-symmetric*. Os RAIDs nível 5 possuem o melhor desempenho comparativo com os outros níveis, a não ser em relação ao nível 1, para leituras pequenas e grandes e para escritas grandes. Porém, assim como os de nível 4, seu desempenho é ruim nas gravações pequenas pois também utilizam o processo ler-modificar-gravar. Felizmente, os sistemas de arquivos estruturados em *log* eliminam este problema pois agrupam várias requisições pequenas de escrita em um amplo *buffer* para depois mandá-las para o disco de uma só vez (Capítulo 3).

Os RAID são normalmente classificados em níveis que vão do 0 ao 6, mas diversas variações para cada um dos níveis são propostas, de forma a torná-los mais robustos, mais econômicos ou com melhor desempenho. Cada um dos níveis possui uma aplicação específica e escolher o melhor depende do uso e das qualidades que se deseja do sistema de armazenamento [CHE94]. Por exemplo, o sistema HP AutoRAID utiliza um controlador inteligente para distinguir entre os dados mais utilizados e formar uma hierarquia de armazenamento secundário [WIL96]. O primeiro nível consiste de um RAID nível 1, mais rápido, porém menor (devido ao seu custo) enquanto o segundo nível utiliza um RAID nível 5, com maior capacidade. Quando certos dados são bastante utilizados, estes são trazidos, de forma transparente ao sistema de arquivos, ao primeiro nível, o contrário acontecendo com dados que se tornam pouco referenciados.

Com o preço dos discos individuais e da lógica que implementa os controladores abaixando cada vez mais, os RAIDs tornaram-se uma realidade para servidores de arquivos, mesmo em instalações mais modestas. Além disso, os novos padrões de interfaces como o Ultra SCSI, com faixa de passagem de 40 MB/s e o FC-AL (*Fibre Channel Arbitrated Loop*), com até 200MB/s, permitem que RAIDs com dezenas de discos equipem estações de trabalho e servidores de arquivos.

2.3.5 Virtual File System

Com a proliferação dos computadores pessoais e das estações de trabalho, os sistemas computacionais tornaram-se muito mais heterogêneos, tanto em relação ao *hardware* quanto ao *software*. Além disso, a necessidade de transferência de dados entre computadores ou redes de computadores torna-se cada vez maior. Assim, todos os sistemas operacionais modernos devem possuir mecanismos para interpretar dados existentes em dispositivos de armazenamento, tais como discos rígidos e disquetes, provenientes de outros sistemas. Entre as diversas variantes do sistema UNIX este problema é muito comum. Geralmente cada fornecedor adota um sistema de arquivos diferente dos demais. Nos UNIX mais antigos, embora fosse permitido que vários sistemas de arquivos fossem montados pelo usuário (na verdade pelo super-usuário) e manipulados pelo núcleo do sistema operacional, estes sistemas deveriam ser sempre do mesmo tipo. Isso ocorria porque o sistema suportava apenas um modelo de sistema de arquivos.

Quando a Sun Microsystems implementou o seu sistema de arquivos de rede, o NFS (*Network File System*) [SAN87], esta optou por manter as referências de arquivos remotos igual à dos arquivos locais. Assim, um identificador único para um arquivo, seja ele local ou remoto, é um caminho (*path*) que se inicia no diretório raiz do sistema ou no diretório atual do processo. Deste modo, um sistema de arquivos remoto é montado e torna-se visível na árvore de diretórios como se fosse um sistema de arquivos local. Embora a referência para arquivos seja a mesma, assim como a interface e a semântica das operações, a realização das operações difere bastante entre o NFS e um sistema de arquivos convencional. Quando um processo necessita ter acesso a um arquivo, local ou remoto, este fornece um caminho para o subsistema de arquivos, que deve descobrir a localização física do arquivo e então realizar a operação. Por exemplo, no sistema de arquivos convencional, um requisição `create` vai direto para as rotinas de alocação de blocos e das estruturas auxiliares em memória. Já no NFS, a chamada é encapsulada pelo processo em uma mensagem RPC (*Remote Procedure Call*) e é enviada para o servidor através da rede. O servidor por sua vez interpreta a chamada, a realiza no sistema de arquivos requisitado e devolve o resultado para o processo como retorno da RPC.

Assim, o subsistema de arquivos do SunOS, onde o NFS foi implementado pela primeira vez, foi modificado para suportar tanto o sistema já existente, o UFS, como o NFS. Felizmente esta modificação permite também que o núcleo do sistema operacional suporte diversos outros modelos de sistemas de arquivos, como o DOS-FAT e RFS (*Remote File System*), desde que estes possuam uma semântica compatível com a semântica do sistema de arquivos do UNIX. O mecanismo que permite tal flexibilidade é chamado VFS (*Virtual File System*) e seu princípio de funcionamento é mostrado na Figura 2.2.

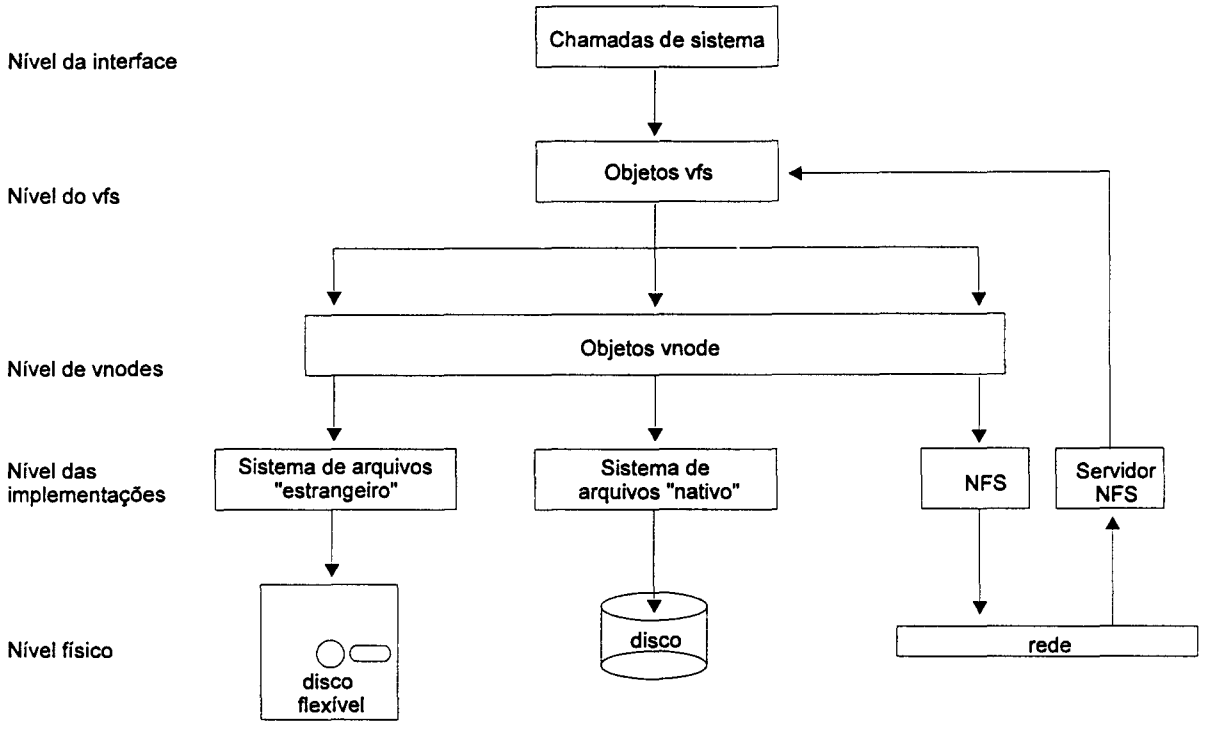


Figura 2.2: O VFS.

O VFS parte do pressuposto de que os sistemas de arquivos UNIX, assim como os arquivos contidos nestes sistemas, devem apresentar características mínimas em comum para que o núcleo e as aplicações possam entendê-los. Por exemplo, todos os sistemas de arquivos no UNIX devem ser "montados" para serem visíveis e os arquivos devem suportar operações mínimas de criação, escrita e leitura, entre outras. Desta forma, o VFS separa as características dos sistemas de arquivos que são dependentes da implementação das que não são dependentes. Para tanto, foram criadas duas classes virtuais para representar os sistemas de arquivos e os arquivos em si. Estas classes são, respectivamente, **`vfs`** e **`vnode`**. As classes do VFS fornecem uma interface padrão pela qual o núcleo do sistema operacional pode ter acesso a implementação de cada um dos sistemas de arquivos.

Cada sistema de arquivos montado é representado no núcleo do sistema operacional como um objeto `vfs`. Os objetos `vfs` formam uma lista ligada cuja cabeça é sempre o sistema de arquivos **raiz** (*root*). A lista é alterada toda vez que um sistema de arquivos é montado ou desmontado da árvore de diretórios.

A classe `vfs` possui informações públicas sobre os sistemas de arquivos, que são utilizadas pelo núcleo do sistema operacional, e informações privadas, que são utilizadas pelas implementações dos sistemas de arquivos. As informações públicas incluem um apontador, chamado `vnodecovered`, para o ponto de montagem deste sistema de arquivos, isto é, o `vnode` que representa o diretório no qual o sistema de arquivos foi montado. As informações

privadas normalmente incluem o superbloco do sistema de arquivos, quando este existe, ou alguma outra informação necessária para manipular as estruturas em disco associadas ao sistema de arquivos. A classe `vfs` ainda possui um apontador para as operações que podem ser realizadas sobre o sistema de arquivos. Estas incluem montar, desmontar, retornar uma referência para o primeiro inode do sistema de arquivos, chamado inode raiz, e recolher informações gerais do sistema de arquivos. A Tabela 2.1 mostra as operações da classe `vfs`.

Um objeto `vnode` representa um arquivo em memória. Assim, cada arquivo aberto possui um objeto `vnode` associado a ele. A classe `vnode` também possui informações públicas e privadas. As informações públicas incluem uma contagem de referências para um objeto `vnode`, que indica quando um `vnode` pode ser apagado da memória, informações sobre *locks* no `vnode`, um apontador para o `vfs` ao qual o `vnode` pertence (`vfsp`) e um apontador para um eventual `vfs` montado no `vnode` (`vfsmountedhere`). O apontador `vfsmountedhere` só possui um valor válido caso o `vnode` seja um ponto de montagem. As informações privadas geralmente correspondem a uma cópia do inode que está no disco. Também existe um apontador para as operações que podem ser realizadas sobre um arquivo, que incluem `open`, `write`, `read`, `link` e `remove`. A Tabela 2.2 traz todas as operações permitidas sobre os `vnodes`.

Como o VFS não faz distinção entre arquivos comuns e diretórios, o `vnode` inclui apontadores para as operações `mkdir`, `rmdir` e `readdir`, aplicáveis apenas a diretórios. Na verdade, quando um certo `vnode` não suporta alguma operação da interface, os apontadores para estas operações devem possuir um valor nulo, ou então o próprio subsistema de arquivos do núcleo deve prover uma função padrão para o `vnode`. O mesmo pode ocorrer com sistemas de arquivos. Um sistema, por exemplo, pode não ser capaz de informar estatísticas sobre si mesmo.

Operação do `vfs`	Descrição
`mount`	Realiza as operações necessárias à implementação para montar um sistema de arquivos. Geralmente lê o superbloco e outras estruturas gerais. Invocada pela chamada de sistema `mount`.
`unmount`	Grava as estruturas modificadas do sistema de arquivos que estão em memória. Invocada pela chamada de sistema `unmount`.
`root`	Retorna um apontador para o `vnode` raiz do sistema de arquivos.
`statfs`	Informações gerais sobre o sistema de arquivos, tais como tamanho do bloco de dados e número de blocos livres para o usuário.
`sync`	Grava os blocos de dados modificados, deixando o sistema de arquivos em um estado consistente.

Tabela 2.1: Operações da classe vfs.

Operação do vnode	Descrição
open	Abre um arquivo.
close	Fecha um arquivo.
rdwr	Lê ou escreve um arquivo.
ioctl	Operações de controle de E/S.
select	Verifica a ocorrência de um evento esperado no arquivo.
getattr	Lê atributos de um arquivo.
setattr	Modifica atributos de um arquivo.
access	Verifica permissão de acesso sobre um arquivo.
lookup	Procura por um nome de arquivo em um diretório.
create	Cria um arquivo.
remove	Remove uma referência (*link*) de arquivo de um diretório.
link	Cria uma referência de arquivo em um diretório.
rename	Renomeia um arquivo.
mkdir	Cria um diretório.
rmdir	Remove um diretório.
readdir	Lê uma entrada de diretório.
symlink	Cria um *link* simbólico.
readlink	Lê o valor de um *link* simbólico.
fsync	Grava os blocos modificados de um arquivo.
inactive	Marca um vnode como inativo.
bmap	Mapeia um endereço de bloco lógico em um endereço físico.
strategy	Lê e grava blocos no sistema de arquivos.
bread	Lê um bloco.
brelse	Libera um bloco na *cache*.

Tabela 2.2: Operações da classe vnode.

A Figura 2.3 representa um sistema com dois tipos de sistemas de arquivos diferentes montados. Para suportar a montagem de vários tipos diferentes de sistemas de arquivos, a chamada de sistema mount foi modificada e recebe três parâmetros: o tipo do sistema de arquivos, o caminho (incluindo o nome do diretório) onde o sistema será montado e um apontador para dados específicos do sistema de arquivos.

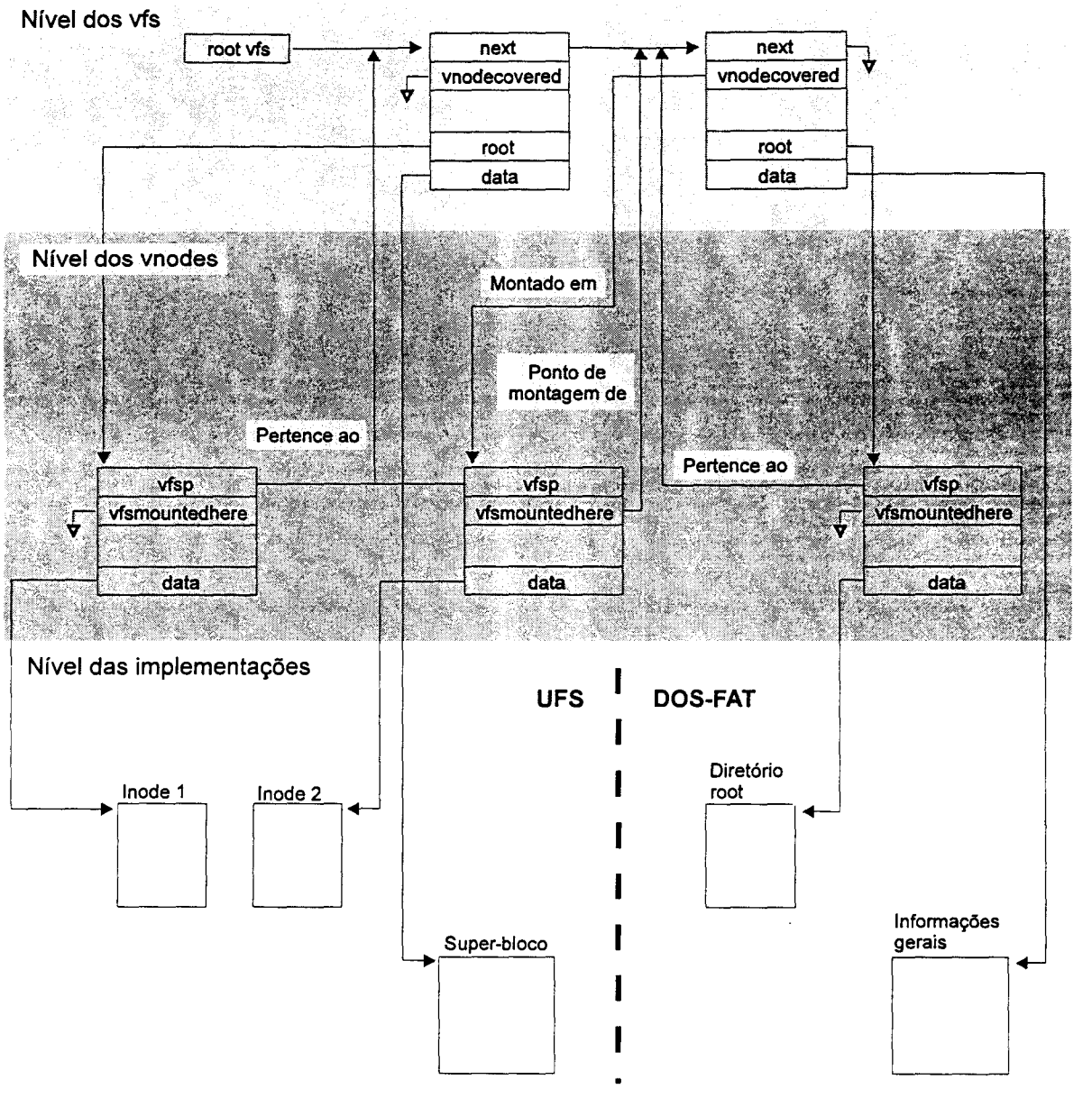


Figura 2.3: Exemplo de uso do VFS.

O primeiro passo da chamada `mount` é criar um objeto `vfs` e encontrar o `vnode` que representa o seu ponto de montagem[2]. Assim, `vnodecovered` do `vfs` aponta para este `vnode` e `vfsmountedhere` no `vnode` passa a apontar para o novo `vfs`. Caso o `vnode` já esteja sendo utilizado como ponto de montagem, `mount` retorna um erro. Em seguida, o

[2] O primeiro sistema de arquivos montado pode ser associado a um `vnode` especial ou a nenhum `vnode`, dependendo da implementação do VFS.

núcleo verifica o tipo do sistema de arquivos, fazendo o apontador de operações do `vfs` apontar para o código das operações da implementação daquele tipo. Só então a operação `mount` do objeto `vfs` é chamada com os dados específicos do sistema de arquivos, que podem incluir o nome de um dispositivo ou uma referência a um servidor remoto, e completa o processo de montagem preenchendo as informações privadas do objeto. O sistema passa então a estar pronto para responder à chamadas aos seus arquivos.

Quando um arquivo é criado, o núcleo do sistema operacional `aloca` um objeto `vnode` para representar em memória o inode do arquivo. O primeiro passo é determinar a qual `vfs` o `vnode` pertence. O núcleo faz `vfsp` do novo `vnode` apontar para o mesmo `vfs` que o diretório no qual está sendo criado. O apontador de operações também é ajustado para as operações específicas do sistema de arquivos para o tipo de `vnode` que está sendo criado (arquivo comum, diretório, *pipe*, *device*, etc.). Por fim, os dados do inode são lidos do disco e a área privada de dados é preenchida. Várias chamadas de sistema que manipulam arquivos recebem um caminho como referência ao arquivo. Assim, o núcleo possui uma rotina de verificação de caminhos, chamada `lookuppn` (*lookup path name)*, que trabalha em conjunto com as implementações dos sistemas de arquivo. Para cada componente do caminho, `lookuppn` chama `lookup` do `vnode` que representa o diretório em questão. A função `lookup` retorna o `vnode` que corresponde ao nome dado dentro do diretório. Este processo termina quando o último componente do caminho é analisado ou quando ocorre um erro causado por algum componente inválido. Se um caminho inicia com "/", o primeiro `vnode` analisado é o `vnode` raiz, caso contrário inicia-se pelo `vnode` do diretório corrente da aplicação.

2.4 UNIX File System

O sistema operacional UNIX possui muitas variantes, que diferem levemente entre si em detalhes de implementação e na semântica de algumas operações. Isso também ocorre com o sistema de arquivos do UNIX, que em cada implementação ou versão nova apresenta uma inovação que o torna mais seguro, acrescenta alguma funcionalidade ou aumenta o seu desempenho. Por motivos didáticos, dividimos o desenvolvimento dos sistemas de arquivos do UNIX em três fases e, para cada fase, será apresentado um sistema de arquivo que a representa melhor. Outras implementações podem diferir em algumas propriedades apresentadas aqui, ou apresentar qualidades não discutidas. O que mais nos interessa é descrever a organização física dos sistemas de arquivos, que é o fator que mais interfere no seu desempenho. Começaremos pelo sistema do System V, do qual derivaram todos os sistemas de arquivos modernos do UNIX. Depois será apresentado o FFS, que aumentou consideravelmente o desempenho do sistema antigo. Por último, será mostrado o UFS da Sun, que incorporou o suporte a diversos tipos de sistemas de arquivos e aprimorou a alocação de blocos do FFS.

2.4.1 System V File System

Embora o sistema original do UNIX System V, o s5fs [VAH96], possuísse características interessantes tais como organização dos dados em diretórios, possibilidade de referência a arquivos por vários nomes (*links*) e integração no espaço de nomes com outros componentes do sistema operacional (como dispositivos e *pipes*), seu desempenho era bastante precário, especialmente com arquivos muito grandes. À medida que os discos foram ficando maiores, os problemas que o s5fs apresentava ficavam mais evidentes.

No s5fs, cada instância do sistema de arquivos ocupa uma partição física do disco e possui um superbloco, colocado no início da partição, que guarda as características da instância. Os arquivos são descritos por estruturas chamadas **inodes**, que guardam atributos tais como dono do arquivo, tamanho, hora da última modificação, permissões de escrita e leitura, além de apontadores para os primeiros blocos de dados do arquivo (normalmente 8), um apontador para um bloco de apontadores indiretos, um para um bloco de apontadores duplamente indiretos e um para um bloco de apontadores triplamente indiretos. Blocos de apontadores indiretos são blocos de índices (endereços) para blocos de dados ou para blocos de apontadores indiretos de nível mais baixo. Por exemplo, blocos de apontadores indiretos possuem os endereços dos blocos de dados indiretos do inode; blocos de apontadores duplamente indiretos possuem endereços de blocos de apontadores indiretos. O tamanho do bloco no s5fs é de 512 bytes e os endereços ocupam 2 bytes. Assim, um bloco de apontadores indiretos possui 128 endereços de blocos. A Figura 2.4 mostra a organização dos blocos de um arquivo.

Diretórios são arquivos especiais cujos registros de tamanho fixo são referências para outros diretórios ou para arquivos comuns[3]. Os principais dados de uma entrada de diretório são o nome do arquivo, que pode ter até 14 caracteres, e o número do inode do arquivo. Assim, um arquivo pode possuir diversos nomes, podendo também estar contido em vários diretórios. Cada nome é dito um *hard link* para o arquivo. Cada inode guarda a contagem do número de referências que existem para si. Assim, um arquivo só é realmente apagado, ou seja, seus blocos de dados são postos na lista de livres e seu inode é liberado, quando a última referência a este for apagada. Não é permitido ciclos na estrutura de diretórios. Deste modo, um subdiretório não pode possuir referência para um diretório de nível superior. Cada sistema possui um inode especial, denominado raiz (*root*), que é o início da árvore de diretórios.

Por causa de limitações nas estruturas internas do sistema de arquivos o número total de inodes no s5fs é de 2^{16}, suficiente para a época na qual foi desenvolvido, mas inaceitável até mesmo para sistemas pessoais atuais. Todos os inodes são agrupados logo depois do superbloco, antes da região destinada aos dados dos usuários. Assim, para cada escrita realizada em um arquivo, a cabeça do disco tem que se deslocar da região de dados para a região onde estão os inodes. No caso de arquivos alocados no final do disco, este *seek* envolve diversos cilindros, aumentando muito o tempo de resposta da operação.

---

[3] Arquivos comuns aqui são aqueles que não possuem atributo de diretório e podem incluir arquivos ordinários, *pipes* e dispositivos.

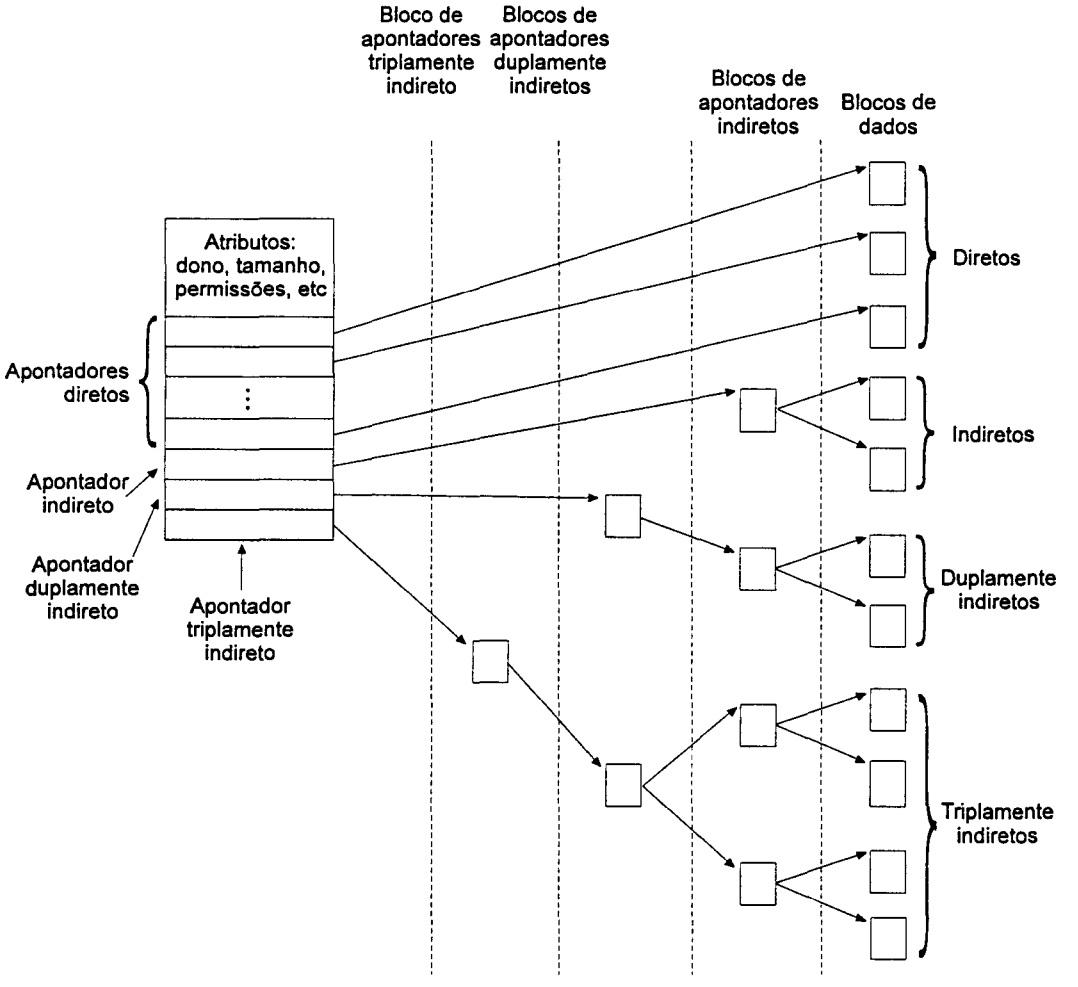


Figura 2.4: Apontadores de blocos de dados de um arquivo no s5fs.

O s5fs utiliza uma lista de blocos livres para localizar novos blocos para serem alocados. Inicialmente, esta lista segue a ordem física dos blocos em disco e todos os arquivos são alocados seqüencialmente. Mas, à medida que arquivos são apagados, expandidos e criados, a lista fica cada vez mais embaralhada e a alocação é feita de forma aleatória.

Devido ao tamanho reduzido do bloco de dados, à separação dos inodes da área de dados e à alocação aleatória, o s5fs só era capaz de utilizar 2% da faixa de passagem (*bandwidth*) do disco. Isso acontecia porque poucos bytes eram transferidos antes que um *seek* fosse necessário. Assim, o tamanho do bloco foi aumentado de 512 bytes para 1024 bytes na Versão 3 do System V. Mesmo dobrando o desempenho do sistema, esta mudança permitiu ao sistema utilizar apenas 4% da faixa de passagem do disco.

2.4.2 Fast File System

Devido principalmente ao baixo desempenho do s5fs, às suas restrições quanto ao número total de inodes e ao tamanho reduzido do nome dos arquivos, além de problemas na segurança e recuperação dos dados, um novo sistema de arquivos, chamado *Fast File System* (ou apenas FFS) [MCK84], foi desenvolvido para o UNIX e incluído no sistema BSD 4.2. Com uma organização física mais aprimorada e algoritmos de alocação mais inteligentes, o FFS consegue taxas de utilização da faixa de passagem do disco entre 20% e 50% [MCV91], dependendo das características do *hardware* utilizado e do *workload* ao qual o sistema é submetido.

A principal meta do FFS era melhorar o desempenho do s5fs sem modificar a semântica do antigo sistema, preservando assim as aplicações que já haviam sido escritas no UNIX e que utilizavam o sistema de arquivos. Desta forma, boa parte das estruturas internas do s5fs foram mantidas. Os arquivos continuam sendo descritos por inodes que guardam os mesmos atributos e apontadores para blocos de dados, blocos indiretos e blocos duplamente indiretos. Porém, o FFS não utiliza o terceiro nível de indireção. Assim, qualquer bloco de um arquivo pode ser encontrado com no máximo três acessos ao disco – um acesso ao inode e dois aos blocos de apontadores – contra quatro no sistema antigo.

Para permitir que arquivos até 4GB[4] pudessem existir mesmo sendo utilizados apenas dois níveis de blocos indiretos, o tamanho do bloco de dados foi aumentado para um mínimo de 4KB ou para qualquer potência de 2 acima disso. Desta forma, além de existir 4 vezes mais dados em cada bloco, cabem 4 vezes mais endereços num mesmo bloco. O aumento do bloco também permite que mais dados sejam transferidos de uma só vez, o que aumenta o *throughput* geral do sistema.

O aumento do bloco de dados causa um efeito preocupante em termos de utilização do espaço do disco. Um arquivo ocupa em média apenas a metade do seu último bloco, assim, o espaço desperdiçado por arquivo aumenta na mesma proporção em que se aumenta o tamanho do bloco. Para solucionar este problema, o FFS permite que o último bloco de um arquivo seja fragmentado, ou seja, ele é dividido em **fragmentos** de mesmo tamanho que podem ser utilizados também por arquivos diferentes. O número de fragmentos por bloco é escolhido no momento da formatação do sistema de arquivos e não pode ser mais modificado. Um bloco pode ser dividido em 2, 4 ou 8 fragmentos, desde que os fragmentos sejam maiores que o bloco físico do dispositivo (geralmente 512 bytes). Se um arquivo necessita de dois ou mais fragmentos, estes devem ser adjacentes dentro do último bloco do arquivo e os demais fragmentos podem ser alocados para outros arquivos. Se um arquivo que possui um bloco fragmentado cresce e necessita de mais fragmentos, o sistema verifica se ainda existem fragmentos adjacentes em número suficiente no mesmo bloco. Se não existir, as informações são copiadas para um outro bloco, o inode é atualizado e os fragmentos do bloco anterior são liberados.

---

[4] Alguns sistemas mais novos utilizam o terceiro nível para permitir a criação de arquivos maiores do que 4GB. Isso depende também do aumento do número de bytes no endereço de blocos.

Embora a organização lógica do s5fs fosse bem elaborada, sua organização física deixava muito a desejar e não levava em conta as particularidades dos dispositivos, como por exemplo o tamanho das trilhas, o número de cilindros e o tempo de posicionamento entre trilhas. Juntamente com a estratégia de alocação dos blocos de dados e a disposição dos metadados no disco, estas características determinam o desempenho dos sistemas de arquivos, pois estão diretamente ligadas ao número de movimentações da cabeça do disco e à latência rotacional média para acessar um ou mais blocos de dados (seção 2.3.1).

Para melhorar a disposição dos metadados no disco, o FFS separa cada partição em diversos **grupos de cilindros**. Cada grupo de cilindros é formado por um ou mais cilindros fisicamente consecutivos no disco e possuem uma organização muito parecida com a do s5fs. Com isso, o FFS tenta tirar proveito da localidade espacial inerente à maioria dos sistemas UNIX, pois cada grupo de cilindros funciona como um "subsistema" dentro do sistema de arquivos. O sistema tenta alocar os dados que possuem algum relacionamento espacial, como um arquivo diretório e os arquivos que estão ligados a ele, num mesmo grupo de cilindros, diminuindo o tempo perdido com *seeks*. Além disso, os blocos dos arquivos são colocados sempre que possível em posições rotacionais ótimas, favorecendo a leitura seqüencial dos dados.

No início de cada grupo de cilindros existe uma cópia do superbloco do sistema. Assim, no caso de perdas de uma ou mais cópias existem outras que podem ser utilizadas. Isso aumenta consideravelmente a segurança em relação ao sistema antigo. A lista de blocos livres foi trocada por um **mapa de bits**, que possui um bit para cada fragmento existente no grupo de cilindros. Sendo **n** o número de fragmentos por bloco do sistema, cada bloco é representado no mapa de bits por um grupo de **n** bits adjacentes, iniciado em uma fronteira de **n** bits. Assim, o sistema sabe exatamente quais blocos estão inteiramente ou parcialmente livres e pode alocar espaço para os arquivos de uma forma bem mais otimizada. Em cada grupo de cilindros existe um número fixo de inodes, posicionados logo após o mapa de bits. Existe um inode para cada 2KB de dados, o que geralmente é uma estimativa acima da média do tamanho dos arquivos em instalações comuns.

Para evitar que todos os superblocos fiquem posicionados na mesma superfície de um único disco, os metadados são colocados em deslocamentos diferentes desde o início de cada grupo de cilindros. Assim, os metadados são colocados em espiral pela unidade de armazenamento, impedindo que danos de um prato ou cilindro causem a perda de todos os superblocos. A região entre o início do grupo de cilindros e o superbloco é utilizada para dados do usuário.

A alocação de blocos era um dos problemas do s5fs que mais influenciavam no seu desempenho. Mesmo quando os arquivos eram alocados contiguamente no disco, a leitura seqüencial muitas vezes não conseguia utilizar nem 10% da faixa de passagem do disco. Isso ocorria porque sistemas com processadores mais lentos perdiam uma revolução a cada leitura de bloco, uma vez que o processador não conseguia tratar as requisições e transferir os dados para a aplicação antes da cabeça atingir o início do próximo bloco. Para solucionar este problema, o FFS permite ao usuário interferir na maneira como o sistema de arquivos aloca os blocos dos arquivos, através do parâmetro **`rotdelay`**. Este pode ser alterado mesmo com o sistema de arquivos já em uso e corresponde ao tempo máximo que o sistema operacional

necessita para responder a qualquer requisição de E/S. Dividindo `rotdelay` pelo tempo gasto para percorrer um bloco, o sistema descobre por quantos blocos a cabeça do disco passa antes que o sistema esteja pronto para uma nova requisição. Assim, a alocação dos blocos é feita de forma otimizada para as características de cada dispositivo de armazenamento.

Cada grupo de cilindros possui ainda um sumário de utilização dos blocos de dados e inodes. Neste sumário encontram-se informações como o número de blocos livres, o número de inodes alocados para diretórios e o número de inodes alocados para arquivos comuns.

As estratégias de alocação dos blocos também envolvem os grupos de cilindros. Como já foi mencionado, o sistema tenta alocar todos os inodes dos arquivos pertencentes a um diretório no mesmo grupo de cilindros que o inode do diretório está. Com isso, o sistema tenta aumentar o desempenho em operações que dependem da localidade espacial, como a listagem de um diretório ou uma aplicação que manipula diversos arquivos em um mesmo diretório antes de passar para outro. Já quando um novo diretório é criado, o sistema tenta encontrar o grupo de cilindros que possui o menor número de diretórios já alocados e o maior número de inodes livres. Assim, sobra espaço para os diretórios crescerem em todos os grupos de cilindros e a estratégia de agrupamento de inodes de um mesmo diretório quase sempre poderá ser realizada.

Os arquivos são alocados nos grupos de cilindros procurando-se o próximo inode livre. Os blocos de dados de um arquivo serão alocados no mesmo grupo de cilindros que o inode do arquivo, até que este atinja um tamanho de 48KB, o que corresponde ao último bloco direto do inode. Depois disso os blocos serão alocados no grupo de cilindros que possua o maior número de blocos livres. O alocador também mudará de grupo de cilindros toda vez que o arquivo aumentar mais 1MB. Isso é feito para evitar que alguns poucos arquivos tomem todo o espaço livre de um grupo de cilindros.

Embora o sistema tente sempre manter os arquivos de forma seqüencial, nem sempre é possível encontrar posições rotacionais favoráveis para acrescentar blocos aos arquivos. Isso ocorre porque o espaço livre do disco tende a ficar mais e mais fragmentado, devido às modificações realizadas nos arquivos. Este problema agrava-se em sistemas que já estão em uso freqüente por alguns meses e nos quais o disco possui utilização acima de 90%, nos quais o desempenho pode cair pela metade [MCK84]. Assim, para poder realizar um bom trabalho na alocação dos blocos na maior parte do tempo, o sistema limita o uso do disco geralmente em 90%. Isso por um lado acaba ajudando aos programas de recuperação de falhas e "defragmentadores", mas aumenta o custo do sistema de arquivos, uma vez que caberão menos dados em cada unidade de armazenamento.

O FFS também acrescentou algumas novas funcionalidades ao sistema de arquivos:

- Os nomes de arquivos podem ter até 255 caracteres. Para não perder espaço nos diretórios com nomes de arquivos pequenos, as entradas possuem tamanho variável (diferentemente do s5fs).
- Foi acrescentada a chamada `rename` à interface do sistema de arquivos, para que a operação de renomear um arquivo fosse atômica. Assim, é assegurado que se a referência ao novo nome de arquivo existir, esta será válida.

- O sistema permite a criação de ***symbolic links*** para os arquivos, além dos ***hard links*** já existentes. Um *link* simbólico é um arquivo que guarda um caminho (*path*) para outro arquivo, ou melhor, para uma referência de diretório a um arquivo. Diferente do *hard link*, a existência de um *link* simbólico não assegura a existência de um arquivo com este nome. A vantagem dos *links* simbólicos é que estes podem referenciar arquivos em outras instâncias de sistemas de arquivos, inclusive de outros modelos e em máquinas diferentes. Novas chamadas de sistema foram implementadas para detectar, escrever e ler os *links* simbólicos.
- Um sistema de *lock* em arquivos foi implementado, substituindo o antigo processo de verificar a existência de um arquivo auxiliar para saber se o arquivo alvo estava sendo utilizado. O sistema trabalha com *advisory locks* e fica a cargo das aplicações decidirem cooperar ou não no seu uso. Os *locks* podem ser exclusivos (apenas um por arquivo em um certo intervalo de tempo) ou compartilhados (vários são aceitos ao mesmo tempo).
- Embora a filosofia do UNIX seja a de compartilhar os recursos do sistema, em alguns casos é necessário um controle mais rigoroso sobre a quantidade de recursos que cada usuário utiliza, especialmente no que diz respeito a disco. Assim, o FFS implementa um mecanismo de quotas, que permite limitar a quantidade de inodes e de blocos alocados por usuário.

Muitos dados dos sistemas de arquivos do UNIX são voláteis e grande parte dos arquivos criados são apagados em menos de um minuto [OUS85]. O tempo de acesso a disco gasto nas operações de um arquivo temporário, que é criado e rapidamente apagado, é desperdiçado pelo sistema. Além disso, o trabalho do sistema de arquivos é dobrado, pois os metadados do sistema necessitam ser atualizados duas vezes em um curto espaço de tempo, uma vez para a criação do arquivo e gravação do blocos de dados e outra para a sua remoção. Assim, o FFS geralmente não realiza a gravação dos blocos de dados de um arquivo antes que se passem 30 segundos. Este comportamento pode ser evitado pelas chamadas de sistema `fsync`, que força a gravação de todos os blocos modificados de um arquivo no disco, e `sync`, que força a gravação de todos os arquivos modificados e todas as estruturas auxiliares em memória de um sistema de arquivos montado. Outro caso onde o FFS grava blocos de dados antes dos 30s terem decorrido é quando a *cache* de blocos se enche.

Infelizmente, para manter a semântica das operações de diretórios e para favorecer a recuperação após falhas do sistema, o FFS continua realizando todas as operações sobre metadados de forma síncrona, ou seja, é assegurado que uma operação que envolva metadados só retorne quando estes metadados estiverem seguros em disco. Operações sobre metadados compreendem operações em arquivos diretório e blocos de apontadores dos inodes.

2.4.3 UNIX File System

Embora o FFS original tivesse um desempenho muito superior ao s5fs, a utilização da faixa de passagem do disco neste sistema não passava de 50%. Este fenômeno acontecia por um problema causado pelo parâmetro `rotdelay`. Quando `rotdelay` é diferente de 0, o número máximo de blocos que podem ser transferidos por revolução é no máximo a metade do número de blocos de uma trilha. Assim, a faixa de passagem efetiva fica reduzida no mínimo à metade.

Para solucionar o problema acima e melhorar ainda mais o desempenho do FFS em acessos seqüenciais a arquivos, foram realizadas algumas mudanças nos algoritmos de alocação que não interferiram nas estruturas do sistema de arquivos nem na sua interface, mas fizeram o sistema possuir bom desempenho tanto em escritas como em leituras e comportar-se como um sistema baseado em extensões [MCV91].

Inicialmente, o *UNIX File System*, ou UFS, era uma implementação do FFS original de Berkeley feita pela Sun Microsystems para o seu sistema operacional SunOS. O UFS já possuía algumas características avançadas como a integração com a camada VFS (seção 2.3.5), que permitia ao sistema operacional suportar outros modelos de sistemas de arquivos, tais como o NFS e o DOS-FAT. O VFS por sua vez era integrado ao sistema de memória virtual do sistema operacional, permitindo que dados de operações de E/S fossem endereçados e tratados da mesma forma que páginas de dados dos processos.

O UFS possui um algoritmo de leitura antecipada de blocos (*read ahead*). Se o bloco que está sendo lido é consecutivo ao bloco anteriormente lido, o sistema interpreta isso como um acesso seqüencial e lê o próximo bloco do arquivo assincronamente, colocando-o em *cache* para futuras referências. A decisão de disparar a leitura antecipada é tomada com referência ao número lógico do bloco do arquivo. O `vnode`, que é a cópia em memória de um inode, possui um atributo chamado `nextr`, que indica qual bloco deveria ser lido para que fosse mantido o acesso seqüencial ao arquivo. Quando o `vnode` é criado, `nextr` recebe 0, indicando que o primeiro bloco do arquivo deveria ser lido. Esta heurística funciona na maior parte dos casos. Se esta previsão for satisfeita, o sistema dispara a leitura antecipada assíncrona do bloco 1 e faz `nextr` ser igual a 1. Este procedimento continua até que um bloco diferente de `nextr` seja requisitado. Quando isso acontece, `nextr` recebe o valor do último bloco lido mais 1. Suponhamos que `nextr` fosse 2 e o bloco requisitado tenha sido o de número 3. O sistema então faz `nextr` igual a 4, mas não dispara a leitura antecipada até que o bloco 4 seja requisitado.

Utilizando a leitura antecipada, o sistema pode efetuar a maioria das requisições de leituras seqüenciais de forma assíncrona. Deste modo, há bastante tempo para o sistema esperar o posicionamento da cabeça do disco, o que torna o parâmetro `rotdelay` desnecessário. Além disso, eliminar `rotdelay` significa que o sistema pode aproveitar melhor os discos que possuem *buffers* de trilhas internos. Porém, no caso de escritas, o sistema não pode fazer o mesmo e continua necessitando de `rotdelay`.

Para eliminar `rotdelay` e conseguir utilizar toda a faixa de passagem do disco, o UFS utiliza uma estratégia diferente para transferir blocos de arquivos. Enquanto o FFS transferia

blocos entre a memória e o disco um de cada vez, o UFS transfere blocos em agrupamentos (*clusters*), ordenando-os na memória antes de gravá-los efetivamente. O tamanho do agrupamento é escolhido de forma que os componentes do sistema de E/S, como controladores de disco e circuitos de DMAs, consigam transferi-lo de uma só vez entre o disco e a memória. O parâmetro do sistema que controla este tamanho é `maxcontig`. Assim, não existe interferência do sistema operacional enquanto os blocos de dados estão sendo transferidos e isto pode ser feito na mesma revolução, mesmo com `rotdelay` sendo 0. O algoritmo responsável pela alocação dos blocos e inodes não foi modificado e as políticas de aproveitamento da localidade espacial e balanceamento de espaço entre os grupos de cilindros continuam sendo as mesmas utilizadas no FFS.

Como o FFS, o UFS também sofre o problema da fragmentação do espaço livre no sistema de arquivos. Embora o sistema tente transferir sempre blocos em agrupamentos de tamanho `maxcontig`, nem sempre isso é possível, devido à fragmentação do espaço em disco (seção 2.4.2). Assim, o sistema é forçado a gravar agrupamentos de tamanho menor que `maxcontig`.

Com a leitura antecipada e a alocação de arquivos em agrupamentos, o sistema consegue tirar proveito de padrões de acesso seqüenciais tanto de escrita quanto de leitura e o desempenho geral do sistema é aproximadamente o dobro do antigo FFS [MCV91], o que de certa forma é intuitivo, uma vez que a utilização da faixa de passagem do disco foi dobrada. Porém, o sistema ainda sofre com acessos aleatórios, pois sempre que um bloco de dados de um arquivo é modificado seu inode necessita refletir esta modificação. Infelizmente, a maior parte dos arquivos em sistemas UNIX são pequenos, com geralmente um ou dois blocos apenas, ou seja, as estratégias de leitura antecipada e *clustering* nada podem fazer para este tipo de *workload*. Assim, a separação física entre inodes e dados é o grande problema do FFS – e das outras implementações que seguem este modelo – pois força a movimentação freqüente da cabeça do disco, aumentando o tempo de transferência dos dados.

Capítulo 3

Sistemas de arquivos estruturados em log

O conceito de *log*[5] é uma técnica largamente utilizada na computação. De forma geral, um *log* pode ser considerado uma estrutura continuamente crescente, onde são armazenados dados, metadados ou informações sobre as modificações realizadas sobre estes dados e metadados. Novas informações são sempre inseridas no final do *log*. Conforme o uso a que se destina, combinações dos três tipos de informação acima também podem ser encontradas em um *log*. Ferramentas de auditoria, contabilização e segurança do sistema utilizam *log* para registrar tentativas de acessos de usuários, falhas do sistema e outros eventos. O *log* é utilizado como armazenamento de dados de usuários especialmente em bancos de dados e sistemas de arquivos.

Alguns bancos de dados utilizam um ou mais *logs* para guardar as modificações realizadas sobre os dados, enquanto estes últimos permanecem intactos em uma região separada de armazenamento. O uso de *log* protege os dados permanentes de falhas durante a gravação e facilita a recuperação após falhas do sistema, pois o único local que necessita ser analisado é o final do *log* (onde as modificações foram realizadas). Este mecanismo ainda contribui para a implementação de transações atômicas. As modificações que fazem parte de uma transação atômica são guardadas no *log* e os dados da área permanente só são atualizados quando a transação for completada com sucesso.

Sistemas de arquivos também podem utilizar um *log* como uma região auxiliar para aumentar a confiabilidade dos dados e aumentar o desempenho nas escritas. Tal técnica é geralmente chamada *journaling* [KAZ90, VER96, SIL98]. Por exemplo, para manter a consistência dos dados, a semântica do FFS (seção 2.4.2) obriga que as operações sobre metadados, em especial sobre arquivos diretório, sejam síncronas. Certos sistemas realizam todas as operações assincronamente e utilizam um *log* para informar quais seriam síncronas. O *log* pode permanecer em memória até que os dados sejam mandados para o disco, o que leva 30s no FFS, por exemplo. Assim, o sistema pode escolher um período em que o disco não esteja sendo utilizado para gravar o *log*. Isso diminui o tempo médio de espera das

[5] A tradução mais próxima: registro, não expressa corretamente a noção de *log* utilizada aqui. Por isso, resolvemos usar a palavra em Inglês, sem tradução.

aplicações, que não precisam aguardar pelo retorno das operações que envolvem diretórios. Além disso, com a técnica de *journaling*, vários acessos ao disco podem ser economizados. Por exemplo, em um sistema de arquivos que não utiliza *log*, como o FFS original, a gravação de dois ou mais blocos de dados indiretos[6] de um mesmo arquivo força a leitura e a rescrita dos blocos de apontadores do inode várias vezes, pois esta é considerada uma operação síncrona. Já com o uso do *log*, estas operações podem ser agrupadas e realizadas de uma só vez no bloco indireto do inode.

Embora um *log* seja logicamente sempre crescente, a certa altura o espaço reservado para este se enche e os dados nele contidos necessitam ser copiados para uma outra região por algum mecanismo do sistema. Geralmente, os dados que ainda são válidos, ou seja, aqueles que não possuem outra referência mais nova no *log*, são copiados para a região permanente de armazenamento e o *log* pode ser então reutilizado (seção 3.1.4). Este processo de compactação do *log* pode degradar bastante o desempenho do sistema e causar esperas prolongadas nas aplicações.

Um dos mais estudados e inovadores modelos de sistema de arquivos dos últimos anos é o *Log-structured File System* (LFS), cujas bases foram propostas inicialmente em [OUS88]. Este é considerado o primeiro trabalho no qual o *log* é a estrutura principal de armazenamento do sistema de arquivos. Isto implica que não existe cópia dos dados em qualquer outro lugar. O primeiro sistema deste tipo realmente viável em termos de desempenho e confiabilidade foi desenvolvido e implementado como parte do projeto Sprite – um sistema operacional distribuído desenvolvido na University of California em Berkeley – por Mendel Rosenblum [ROS92b].

Diversos outros trabalhos desenvolvidos posteriormente [SEL92, BUR92, HAR95, DEC96], utilizam a mesma filosofia do Sprite LFS e diferem pouco em sua estrutura interna. Neste capítulo serão apresentados os princípios do sistema precursor e as modificações propostas por outros trabalhos. Ao final, será apresentada uma análise sobre o que nós consideramos as melhores características do LFS e as que ainda podem ser melhoradas.

3.1 Sprite LFS

O LFS compartilha a idéia de vários sistemas de arquivos modernos, dentre os quais o FFS, de que as operações de leitura podem ser atendidas muito mais rapidamente se os dados estiverem em uma *cache* na memória principal. Assim, o LFS possui uma organização física dos dados que otimiza as operações de escrita dos arquivos, sejam estas seqüenciais ou não, e pressupõe que o sistema conta com uma *cache* de blocos eficiente para atender às solicitações de leitura.

A idéia fundamental do LFS é coletar diversas operações de escrita em memória principal, organizar estas operações de forma que os blocos de um mesmo arquivo sejam gravados seqüencialmente no disco e salvar o resultado destas operações de uma só vez. A

[6] Blocos com índice maior que o número de apontadores diretos do inode.

estrutura principal de armazenamento do LFS é o *log*, que cresce continuamente conforme as operações são acrescentadas no seu final.

Utilizando a gravação seqüencial, o LFS consegue um desempenho até dez vezes maior que o FFS para escritas de arquivos pequenos, onde grande parte do tempo é gasto na gravação dos metadados dos arquivos [ROS92a, SEL95]. O grande aumento de desempenho deve-se ao fato de que a organização dos sistemas de arquivos baseados no FFS obriga uma movimentação exagerada da cabeça de leitura/gravação do disco em arquivos pequenos. Esta movimentação é necessária para atualizar os inodes destes arquivos, que ficam em uma região separada dos blocos de dados. Além do LFS coletar várias requisições antes de mandá-las para o disco e gravá-las seqüencialmente, os metadados são gravados juntamente com os blocos de dados dos arquivos. Assim, apenas uma movimentação é necessária para que todas as informações sejam salvas. O LFS ainda possui um desempenho comparável ao FFS em quase todos os padrões de acesso, a não ser na gravação aleatória e posterior recuperação seqüencial de arquivos com muitos blocos de dados. Felizmente, este padrão de acesso não é comum em ambientes de engenharia, de escritórios e educacionais (seção 2.2), para os quais o LFS foi desenvolvido.

Um problema atribuído ao uso de *log* é a queda de desempenho do sistema devido à limpeza, ou compactação, que este necessita. Quando feita de forma concorrente com as operações normais do sistema de arquivos, a limpeza do *log* pode degradar o desempenho geral do sistema de arquivos em até 40%, dependendo do *workload* ao qual o sistema é submetido [SEL93b]. Mas heurísticas mostram que na grande maioria das vezes o processo de limpeza pode ser disparado sem contenção com as operações normais do sistema de arquivos (seção 3.2.6).

3.1.1 Organização do log

Um dos objetivos do LFS foi manter a mesma semântica do sistema de arquivos convencional do UNIX, o FFS, e compatibilidade com a sua interface. Por isso, foram mantidas as estruturas fundamentais do FFS, como o **superbloco**, os **inodes** (descritores de arquivos) com apontadores para **blocos de dados, blocos indiretos** e **blocos duplamente indiretos**, e os **arquivos diretório** (seção 2.4.1).

Não existe no LFS a separação do disco em grupos de cilindros, que é a principal inovação proposta pelo FFS. Ao contrário disso, o disco é dividido em extensões de mesmo tamanho, dispostas fisicamente em seqüência, chamadas **segmentos**. O encadeamento dos segmentos formam o *log* mas, devido ao processo de limpeza (seção 3.1.4), a ordem lógica dos segmentos não corresponde à sua ordem física.

O sistema tenta gravar informações no *log* sempre em rajadas do tamanho de um segmento, mas nem sempre isso é possível e certos segmentos são parcialmente preenchidos. As gravações parciais de segmentos dependem das políticas de gravação de blocos do LFS e das requisições das aplicações. Por exemplo, uma aplicação pode requisitar uma operação `fsync` em um arquivo, forçando os seus blocos a serem escritos no disco. As políticas do LFS

asseguram que os blocos de um arquivo serão salvos no disco no máximo 30s após serem criados ou modificados ou, caso uma requisição `fsync` tenha sido feita sobre o arquivo, antes que esta retorne para a aplicação. Assim, se existem blocos de dados em *cache* que foram modificados há 30s, estes blocos serão salvos para o *log*, mesmo que não ocupem um segmento inteiro. Além dos blocos de dados, são salvos o inode do arquivo ao qual os blocos pertencem e seus blocos indiretos de apontadores.

O tamanho dos segmentos é tipicamente 512KB, mas isso pode variar de instância para instância do sistema de arquivos. O que importa é que o tamanho dos segmentos deve ser grande o bastante para que o volume de dados transferidos na gravação compense o tempo de posicionamento da cabeça de leitura/gravação. À primeira vista, pode parecer que as gravações parciais ferem a estratégia de gravar grandes extensões sem necessidade de posicionar a cabeça do disco novamente. Porém, isso na verdade não ocorre. Em primeiro lugar, mesmo com gravações parciais, todas as informações são inseridas no final do *log*. Assim, a cabeça do disco deve permanecer na mesma trilha onde estava antes da gravação ser efetuada, ou numa trilha adjacente, dependendo do tamanho do segmento e da geometria do disco. Felizmente, o tempo de posicionamento trilha a trilha é baixo e não acarreta perdas de desempenho expressivas. O pior que poderia ocorrer é que a próxima gravação parcial deveria esperar uma revolução para que o final do *log* fosse novamente alcançado. Porém, se o sistema não conseguiu gerar informações suficientes para encher um segmento em 30s, é bem provável que este esteja sendo pouco utilizado no momento e esperar por uma revolução não seria grande problema para as aplicações (especialmente em ambientes de escritório, engenharia e ensino).

Outra estrutura nova no LFS é a **região de *checkpoint***, onde se encontram algumas informações sobre o estado do sistema de arquivos. Existem duas regiões de *checkpoint* em locais fixos no disco, cada uma com uma marca de tempo (*timestamp*) e um apontador para o fim do *log*. Estas são gravadas periodicamente e de forma alternada. Quando o sistema é montado, ambas as regiões são lidas e é utilizada aquela que possuir a marca de tempo mais recente. Este mecanismo permite a recuperação de falhas ocorridas durante a gravação do *checkpoint*. A Figura 3.1 mostra a organização lógica do Sprite LFS. O *checkpoint* determina um instante no qual todas as estruturas do sistema de arquivos estão consistentes.

3.1.2 Mapa de inodes

Uma diferença radical do LFS sobre o FFS é que os inodes não se localizam em uma região fixa do disco, estando distribuídos pelo *log*. Assim, para que estes sejam encontrados facilmente, existe outra estrutura chamada **mapa de inodes**, que é uma tabela indexada pelo número do inode. A Figura 3.2 é um exemplo da criação de dois novos arquivos, `dir1/file1` e `dir2/file2`, cada um com apenas um bloco de dados. A organização mostrada na figura é lógica, pois na verdade os inodes encontram-se agrupados em blocos, como veremos abaixo.

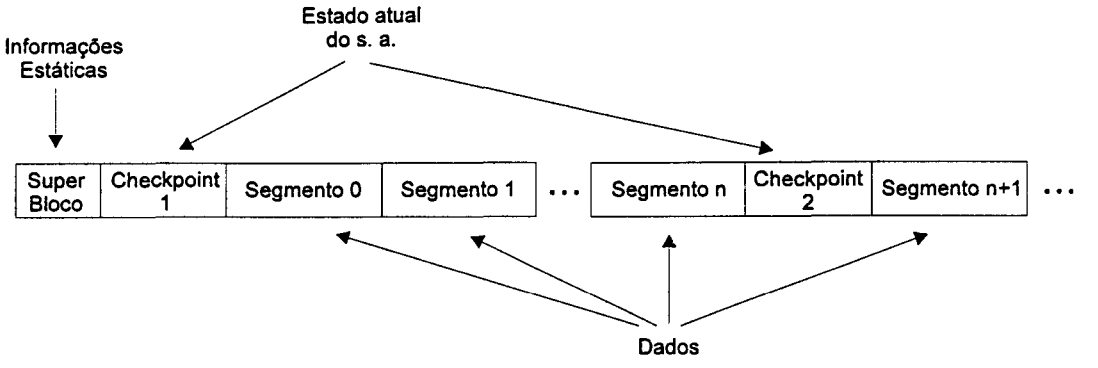


Figura 3.1: Organização lógica do Sprite LFS.

Assim como no FFS, a unidade mínima de alocação do LFS é um bloco, de forma que o LFS divide os segmentos em blocos. Sendo assim, um inode, quando gravado, ocuparia o espaço de um bloco, que geralmente está entre 1KB e 8KB. Como um inode possui pouco mais que uma centena de bytes, o espaço desperdiçado seria muito grande. Por isso, o LFS agrupa diversos inodes em blocos lógicos do sistema de arquivos, os quais recebem o nome de **blocos de inodes**. Este procedimento, além de economizar espaço em disco, também permite que vários inodes sejam carregados para a memória com uma só operação de E/S, o que favorece o desempenho do sistema.

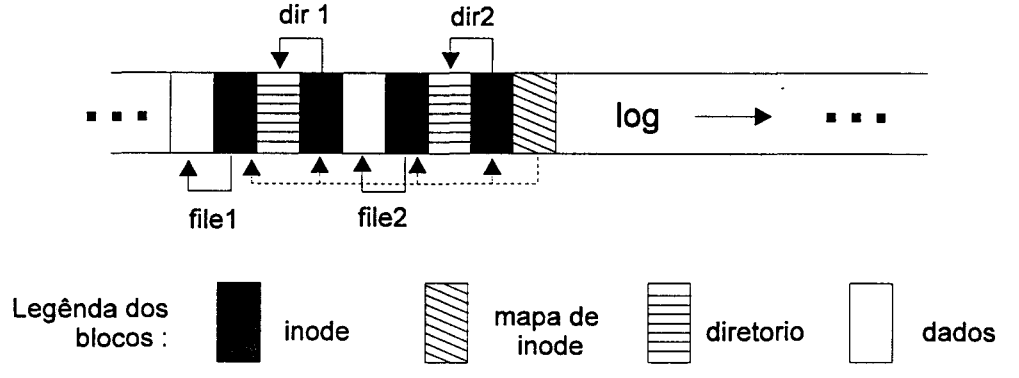


Figura 3.2: Criação de dois arquivos no LFS.

O desempenho do LFS depende da gravação seqüencial dos dados. Deslocar a cabeça do disco para realizar atualizações em blocos distantes do final do *log* pode levar por terra todo o benefício desta organização. Assim, o LFS possui uma política de não rescrita dos dados. Quando um arquivo é alterado, os blocos necessários são lidos para a memória principal, alterados e colocados no *buffer*. Estes blocos serão gravados no final do *log*, juntamente com outros vindos das demais operações de E/S. Desta forma, serão associados novos endereços físicos para os blocos de dados e os apontadores do inode do arquivo devem ser atualizados. Até mesmo os metadados seguem esta política. Por isso, qualquer alteração nos atributos de um arquivo causam a mudança de posição do inode que o representa.

O mapa de inodes também é dividido em blocos, cada qual possuindo diversas entradas. Estes blocos, assim como os demais blocos de metadados, são gravados no *log*. Note que o número de entradas do mapa de inodes por bloco é bem maior que o número de inodes por bloco, pois cada linha do mapa só possui o número do bloco no qual o inode se encontra, sua posição dentro deste bloco, o seu número de versão e a hora do último acesso ao inode. A hora do último acesso é mantida no mapa de inodes por razões de desempenho. Se fosse mantida no inode, todas as vezes que este fosse lido, mesmo sem ser modificado, a hora de acesso teria que ser atualizada, forçando uma nova gravação do inode. Assim, a cada acesso, o inode ficaria mais longe de seus dados e as suas cópias desatualizadas ocupariam espaço desnecessário em disco. O número de versão será descrito adiante.

Da mesma forma que o mapa de inodes existe para identificar rapidamente a posição de cada inode no *log*, existe um vetor, localizado na região de *checkpoint*, que indica a posição de cada bloco do mapa de inodes. Quando o sistema é montado, esta tabela é carregada para a memória juntamente com o **superbloco** e permanece ali até que este seja desmontado. A Figura 3.3 traz a seqüência realizada para se encontrar um inode qualquer, no caso o de número 1020. Considerando que cada bloco do mapa de inodes possui 100 entradas, o endereço do bloco de inodes onde o inode se encontra está no bloco 10 do mapa de inodes. Pode-se observar que os blocos de inodes não são indexados pelo número do inode, ou seja, a posição do inode é determinada apenas na hora de sua criação ou em modificações posteriores.

3.1.3 Segmentos

Os segmentos possuem uma organização interna que possibilita ao sistema recuperar a consistência dos dados em caso de falha. Todo segmento utilizado possui ao menos um **bloco de sumário**, que é um descritor dos dados do segmento. Para cada bloco de dados do segmento, o sumário possui o número e a versão do inode do arquivo ao qual esse bloco pertence e a sua posição lógica dentro do inode, isto é, o número lógico do bloco no arquivo representado pelo inode. O número de versão do inode é incrementado toda vez que o arquivo que este representa é truncado ou quando o arquivo é apagado.

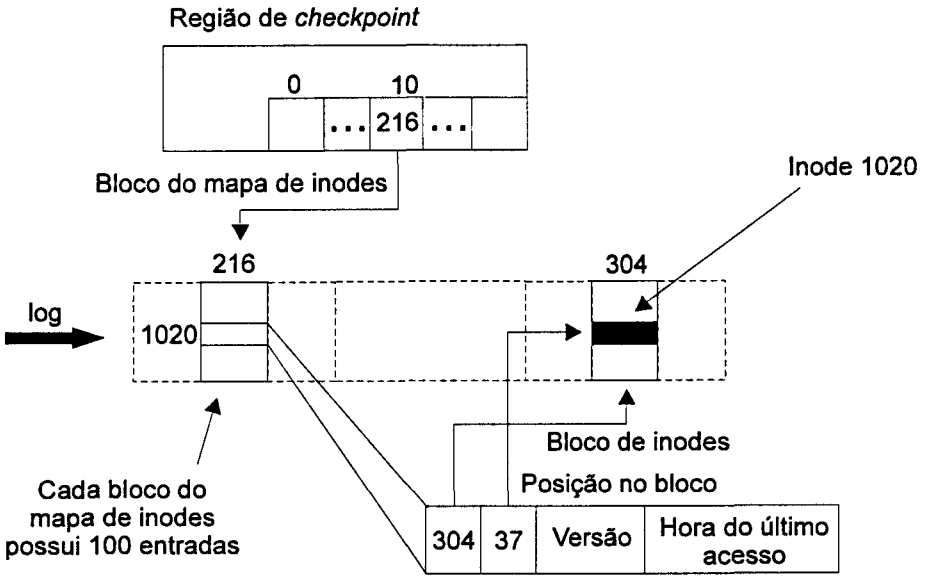


Figura 3.3: Como encontrar um inode no log.

O sumário ainda possui, em seu cabeçalho, apontadores para os segmentos logicamente anterior e posterior no *log* e um apontador para o próximo bloco de sumário dentro do segmento. Assim, uma ferramenta de recuperação pode atravessar o *log* em ambos os sentidos.

Um segmento pode possuir mais que um bloco de sumário, devido a gravações parciais (seção 3.1.1). Assim, cada sumário descreve uma **região de *log***, que é a unidade de gravação do LFS e que pode compreender desde um único bloco de sumário até um segmento inteiro. O sumário é a última estrutura da região do *log* a ser gravada. Dessa forma, ao encontrar um bloco de sumário o programa de recuperação saberá que os dados gravados anteriormente a esse estão consistentes.

Outra estrutura que auxilia na recuperação de falhas é o **registro de operações de diretório**. É assegurado pelo sistema que esta estrutura seja sempre gravada no segmento antes de qualquer operação que envolva diretórios, como a criação e a remoção de arquivos. Isso permite que a informação sobre a operação esteja no disco antes que os elementos que a compõem (o inode ou o bloco do diretório). No caso de algum dos elementos não ser encontrado no segmento, devido à uma falha no momento de sua gravação, o programa de recuperação saberá qual operação estava sendo realizada e, se possível, poderá completá-la. Cada entrada do registro de operações de diretório possui um código de operação (que pode ser `create`, `link`, `unlink` e `rename`), o número do inode do diretório, posição da entrada no diretório, o número do inode do arquivo e seu nome.

3.1.4 Limpeza do log

Por causa da política de não rescrita, o espaço livre em disco no LFS pode se esgotar muito rapidamente. Assim, é necessário um mecanismo coletor de lixo que separe os blocos que possuem informações desatualizadas, ou **blocos mortos**, dos que ainda são válidos, chamados **blocos vivos**. Também é necessário reorganizar e compactar o *log*, de forma a criar segmentos livres onde este possa crescer. Este coletor recebe o nome de ***cleaner***.

Quando uma nova instância do sistema de arquivos é criada, todos os seus segmentos – a não ser possivelmente o primeiro, que possui informações sobre o inode raiz e outros inodes especiais do sistema – estão limpos, ou seja, não possuem nenhum bloco de dados válido. À medida que o sistema requer espaço, um novo segmento é escolhido e marcado como o **segmento atual**, onde serão gravadas as informações do *buffer*. Quando um limite mínimo de segmentos limpos – normalmente determinado na formatação do sistema de arquivos – é alcançado, o processo de limpeza deve ser disparado. A Figura 3.4 exemplifica esta operação. O *cleaner* escolhe um ou mais segmentos para serem limpos, segundo um critério elaborado para otimizar o tempo gasto e o espaço ganho. Apenas os blocos válidos são copiados para o segmento atual e os segmentos analisados são marcados como limpos. O critério de escolha do segmento, como veremos abaixo, é um ponto muito importante no desempenho do sistema e na eficiência do processo de limpeza. Dependendo do segmento escolhido, o espaço livre gerado não compensa o tempo gasto limpando-o.

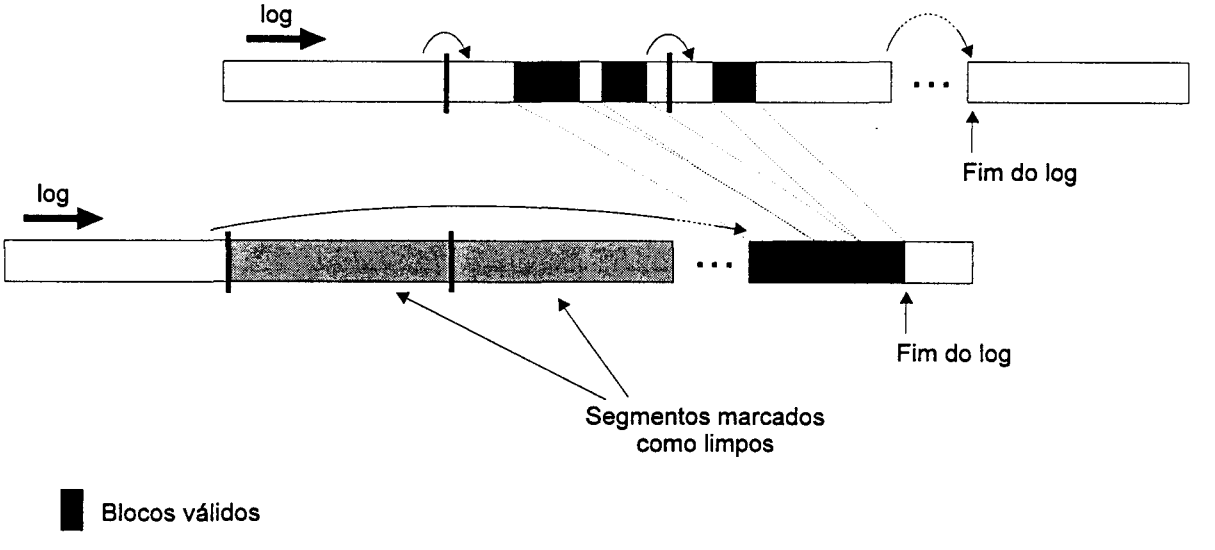


Figura 3.4: Processo de limpeza de segmentos.

Para verificar se um bloco ainda é válido, o sistema olha no sumário do segmento a qual inode esse pertence e qual a sua posição lógica no arquivo. O sistema procura no mapa de inodes a posição do inode no *log*. Antes de ler o inode, o número de versão que existe no sumário do segmento é comparado com o número de versão do mapa de inodes. Se estes forem diferentes, isso indica que o arquivo que o inode representa foi truncado para zero bytes ou que foi apagado e o inode está sendo usado para um outro arquivo. Em ambos os casos, todos os blocos de dados deste inode que estejam no sumário são inválidos e podem ser desprezados e o inode não necessita ser lido. Caso os números de versão sejam os mesmos, o sistema lê o inode e verifica o número do bloco físico que está indicado no apontador referente à posição lógica procurada. Se este número for igual ao número do bloco que está no sumário sendo analisado, o bloco ainda é válido e deve ser gravado novamente; senão, pode ser descartado. Quando um inode é lido, todos os blocos que estão no sumário e que lhe pertencem são analisados.

Quando todos os blocos de sumário de um segmento foram analisados, o segmento é retirado do encadeamento que forma o *log* e é marcado como livre, podendo ser utilizado quando necessário. A quantidade de segmentos limpos em cada invocação do *cleaner* também depende das políticas de limpeza.

A escolha do segmento a ser limpo é um dos fatores que mais influenciam no desempenho geral do LFS. Se a escolha for mal feita, o tempo gasto para limpar o segmento pode causar paradas momentâneas no sistema de arquivos e o espaço livre gerado pode ser pequeno, esgotando-se rapidamente e forçando uma nova invocação do *cleaner*. O custo da limpeza é o tempo gasto para ler o segmento, encontrar os blocos vivos, reordená-los e gravá-los novamente em um novo segmento. Todos estes procedimentos dependem da quantidade de blocos válidos que existem dentro do segmento, embora o que realmente pese é o tempo gasto para leitura e rescrita dos dados. Além disso, a recompensa pelo tempo gasto no processo de limpeza é o espaço liberado pelas informações não válidas. Sendo assim, limpar um segmento que possua menos blocos vivos é muito mais proveitoso, pois gasta-se menos tempo e ganha-se mais espaço. Para saber rapidamente a taxa de utilização dos segmentos, o LFS conta com uma estrutura chamada **tabela de utilização de segmentos**. Esta tabela é dividida em blocos que são salvos no *log* e a região de *checkpoint* possui um vetor de apontadores para estes blocos. Cada entrada da tabela de utilização de segmentos possui o número de bytes válidos e a idade do segmento, ou seja, a hora em que o último bloco de dados foi modificado antes de ser gravado. Um segmento é implicitamente marcado como limpo quando possui zero bytes válidos.

Intuitivamente pode parecer que o melhor segmento a ser escolhido para limpeza é sempre aquele que possui o menor número de blocos válidos. Essa política de limpeza é conhecida como política gulosa (*greedy police*), que foi a primeira política utilizada no *cleaner* do Sprite LFS. Infelizmente, os resultados obtidos com esta política ficaram bem abaixo do esperado. Para entender porque esta política não apresenta bons resultados, devemos analisar o impacto da limpeza no desempenho do LFS.

Uma métrica utilizada para avaliar a eficiência de um sistema de arquivos é o **custo de escrita**. Este é definido como a média de tempo que o disco fica ocupado por byte novo escrito [ROS92b]. Um custo de escrita de 1.0 é ideal e significa que os dados são escritos

utilizando-se toda a faixa de passagem do disco. Um custo de escrita de 10 significa que apenas 10% da faixa de passagem está sendo efetivamente utilizada para novos dados. No FFS, este tempo depende de atrasos como posicionamento da cabeça do disco e latência rotacional para alcançar os setores requisitados. O custo médio do FFS está na ordem de 10 e o custo médio do FFS com utilização de *log* e escritas adiadas está em torno de 4, ou seja, 25% da faixa de passagem do disco.

No LFS, o atraso de posicionamento e a latência rotacional são desprezíveis, mas a certa altura é necessário que seja feita a limpeza dos segmentos, que requer que os segmentos sejam lidos e os dados ainda válidos sejam rescritos no *log*. Assim, o custo de escrita é dado por:

$$custo\ de\ escrita = \frac{bytes\ lidos + bytes\ validos\ escritos + novos\ bytes\ escritos}{novos\ bytes\ escritos}$$

Para cada segmento de novos dados adicionados ao *log*, o espaço livre de um segmento deve ser gerado pelo processo de limpeza. O espaço gerado depende da taxa de utilização, ou fração de utilização, **u**, dos segmentos que foram limpos. Desta forma, **N** segmentos com taxa média de utilização **u** devem ser limpos para que um segmento inteiro seja liberado. Assim, a fórmula acima fica sendo:

$$custo\ de\ escrita = \frac{N + N * u + N * (1-u)}{N * (1-u)} = \frac{2}{1-u} \quad , sendo\ 0 < u < 1 \qquad ①$$

Quando os segmentos lidos possuem taxa de utilização igual a 1, ou seja, estão totalmente cheios, o espaço gerado será zero e podemos dizer que o custo de escrita será infinito. Para uma taxa de utilização igual a 0, ou seja, os segmentos não possuem nenhum dado válido, o custo de escrita será 1, pois os dados não necessitam ser lidos nem rescritos.

Desta forma, o custo de escrita no LFS depende diretamente da taxa de utilização dos segmentos escolhidos para serem limpos, o que difere do FFS, onde o custo de escrita não depende da utilização do disco[7]. A Equação 1 também determina que para o LFS possuir uma eficiência nas escritas maior que a do FFS com escritas adiadas, ou seja, possuir um custo de escrita menor que 4, os segmentos limpos devem ter utilização média menor que 50%. A medida mais simples – e menos inteligente – para alcançar este objetivo seria limitar a taxa de utilização de todos os segmentos do sistema de arquivos em 50%. Porém, o custo por bytes de tal sistema seria quase o dobro de um sistema FFS, que permite utilizar 90% da capacidade do disco. A solução para que o LFS possa realizar a limpeza de segmentos mesmo com altas taxas de utilização do disco e ainda possuir desempenho ao menos comparável com o FFS, é conseguir uma distribuição de dados entre os segmentos de forma que alguns deles estejam quase vazios e a maior parte possua utilização próxima de 100%.

[7] Desde que sistema conte com no mínimo 10% de espaço livre em disco para alocar os blocos dos arquivos em agrupamentos.

Infelizmente, a política gulosa cria um acúmulo de segmentos com taxas de utilização altas. Em estudos realizados no Sprite LFS, para uma utilização média de 75% do espaço total do disco e com todos os arquivos tendo a mesma chance de serem referenciados, os segmentos eram limpos com cerca de 55% dos dados válidos e, pior, existia um acúmulo de segmentos com taxas de utilização pouco acima deste limite mínimo. Isso ocorre porque todos os segmentos vão tendo cada vez mais blocos mortos, mas estes só são limpos quando se tornam o menos utilizado. Assim, diminui a vantagem de se limpar qualquer um dos segmentos, pois o espaço liberado é pequeno e logo será necessário limpar outro segmento.

A política gulosa foi testada também levando-se em conta o padrão de acesso que se observa na maioria dos sistemas multi-usuários. Nestes sistemas, cerca de 10% dos arquivos são modificados de forma constante e recebem 90% das referências. Estes são chamados arquivos "quentes". Os 90% restantes quase nunca são modificados e são chamados "frios". Mais uma vez, contrariando a intuição, o acúmulo de segmentos próximos do ponto de limpeza foi maior que o observado com o padrão de acesso uniforme. Por causa disso, os segmentos passaram a ser limpos com taxas de utilização um pouco mais altas – em torno de 60%.

Também foi constatado que o aumento da localidade de referência piorava cada vez mais a distribuição dos segmentos. A explicação para tal fenômeno é que os segmentos que possuem dados "quentes" têm sua taxa de utilização abaixada muito mais rapidamente e são limpos com maior freqüência que os segmentos que possuem dados "frios". Por outro lado, os segmentos "frios" demoram para serem limpos e o espaço livre que possuem fica um grande tempo ocioso antes que seja aproveitado. Assim, a política gulosa limpava segmentos em que os arquivos ainda tinham chance de serem apagados e utilizava novos segmentos para guardá-los. Esta também não conseguia fazer proveito do espaço livre existente nos segmentos "frios".

A solução para o problema acima foi modificar a política implementada no *cleaner*, de forma a se conseguir uma distribuição onde sempre existissem segmentos com pequenas taxas de utilização que compensassem o custo da limpeza. O critério de escolha de segmentos desta nova política leva em conta a sua idade, além do espaço livre gerado. A idéia é que o custo de limpeza de cada byte depende acima de tudo do tempo que este byte permanecerá válido.

Desta forma, chegou-se à conclusão de que o espaço liberado por um segmento com dados mais estáveis era mais valioso que o espaço liberado por um segmento com dados mais voláteis. No Sprite LFS, a medida utilizada para determinar quão estável é um segmento é a sua idade. A idade de um segmento é dada pela hora em que o seu bloco mais jovem foi modificado pela última vez. Assim, espera-se que segmentos mais antigos possuam menor probabilidade de serem modificados do que um segmento que possui dados mais novos. Assim, a nova política ficou conhecida como **custo-benefício**. Na política de custo-benefício, o segmento escolhido é o que possui menor valor na relação:

$$\frac{espaco\ livre\ gerado\ *\ idade\ dos\ dados}{custo\ da\ limpeza} = \frac{(1-u)*idade}{1+u} \qquad ②$$

sendo **u** a taxa de utilização do segmento.

Com esta nova política, o sistema consegue uma distribuição onde a grande maioria dos segmentos possui entre 70 e 100% de dados válidos e alguns segmentos possuem entre 20 e 40% de dados válidos.

3.1.5 Recuperação pós falha

Como já vimos, para gravar os metadados no *log* juntamente com os dados e para realizar o processo de limpeza, o LFS necessita de estruturas de índice, posicionadas em um lugar fixo no disco chamado região de *checkpoint*. Cada região de *checkpoint*, que no Sprite LFS são duas, possui dois vetores de endereços, um para os blocos do mapa de inodes e um para os blocos da tabela de utilização de segmentos. Estas duas estruturas determinam o estado atual do sistema de arquivos, uma vez que indicam quais são as cópias válidas de cada inode, que por sua vez apontarão para os blocos válidos de cada arquivo, e quais os segmentos que estão sendo usados – e portanto possuem dados válidos. O *checkpoint* ainda guarda outras informações necessárias ao sistema:

- Uma marca de tempo que será utilizada pelo sistema para escolher qual das duas regiões está mais atualizada.
- A localização da última região de *log* gravada e que marca o final do *log*.
- Informações sobre o número de blocos livres no sistema e disponíveis para os usuários.
- Contadores para contabilização do sistema de arquivos.

Para manter o desempenho a níveis aceitáveis, o LFS mantém uma cópia do *checkpoint* em memória, de forma que não é necessário fazer um acesso ao disco toda vez que uma de suas estruturas é modificada. Os vetores e as outras informações não são muito grandes e não ocupam muito espaço da memória principal. Da mesma forma, os blocos do mapa de inodes e da tabela de utilização de segmentos que foram modificados não são prontamente gravados no *log*. Se estas estruturas fossem gravadas toda vez que fossem modificadas, as regiões de *log* teriam muito mais metadados e o espaço desperdiçado seria muito grande.

Em um sistema totalmente livre de falhas, estas cópias atualizadas do *checkpoint*, do mapa de inodes e da tabela de utilização de segmentos só precisariam ser salvas quando o sistema de arquivos recebesse a chamada de sistema `umount`. Porém, caso ocorra uma falha no sistema, como uma queda de energia, o estado atual do sistema se perderá. Assim, todas as estruturas de índice que estão em *cache* são gravadas em intervalos regulares de tempo, cujo valor é guardado no **superbloco** do sistema de arquivos.

Na primeira fase do processo de *checkpoint*, o sistema grava qualquer bloco de dados, bloco de apontadores e bloco de inodes modificados, depois grava os blocos do mapa de inodes e os da tabela de utilização de segmentos. Se mais de uma região de *log* for necessária, o processo prossegue até que o último bloco de sumário de segmento seja gravado. A fase

final consiste na gravação de uma das regiões de *checkpoint*, que possui um apontador para o último sumário gravado e uma marca de tempo atual, marcando o final do *log*.

O intervalo padrão utilizado no Sprite LFS é de 60s. Isso significa que, se alguma falha ocorrer durante os 60s após a última gravação do *checkpoint*, os dados salvos no *log* posteriormente a este correm o risco de serem perdidos. Para evitar que isso ocorra, o LFS conta com um processo de recuperação de falhas, chamado *roll-forward*.

Como as modificações são sempre efetuadas no final do *log*, caso ocorra uma falha, a parte afetada serão as últimas regiões de *log* escritas. O primeiro passo do programa de recuperação do Sprite LFS é encontrar a última região de *checkpoint* salva, que indica o último estado consistente do sistema. Todas as regiões de *log* posteriores ao *checkpoint* estão potencialmente inconsistentes e necessitam ser analisadas pelo procedimento *roll-forward*. As regiões de *log* potencialmente inconsistentes são encontradas pelo encadeamento dos sumários a partir da última região de *log*, indicada na região de *checkpoint*. O processo de *roll-forward* consiste na reconstrução dos metadados do sistema através das informações dos sumários e dos registros de operações de diretório. Por exemplo, se um sumário indica que no segmento existem blocos de dados de um inode mas este não foi gravado no disco, o programa lê o inode, utilizando as últimas estruturas de índice válidas, atualiza-o e atualiza também o mapa de inodes.

Quanto mais espaçado for o período de gravação do *checkpoint*, maior será o trabalho de recuperação de falhas. Porém, cada vez que o *checkpoint* é realizado, a cabeça do disco é movida, muitas vezes para longe do segmento que está sendo gravado. Assim, intervalos pequenos podem degradar muito o desempenho do sistema. Uma alternativa aos intervalos fixos de gravação seria esperar até que uma certa quantidade de dados tivesse sido salva no *log*.

3.2 BSD LFS

O principal trabalho realizado sobre o LFS após o seu desenvolvimento por Rosenblum foi o de Margo Seltzer, em sua tese de Doutoramento [SEL92]. Embora o objetivo principal de seu trabalho fosse implementar suporte a transações atômicas no sistema de arquivos [SEL93b], ela acabou contribuindo para o aprimoramento do LFS e iniciou discussões sobre o seu desempenho que até hoje são motivo de estudos. Suas propostas corrigem certos problemas do projeto original e o tornam mais seguro. Algumas modificações nas estruturas internas do LFS favorecem a implementação de suporte à transações atômicas pelo sistema de arquivos. Outras referem-se ao modo como as informações são realmente gravadas em disco e solucionam alguns erros conceituais que poderiam provocar mau funcionamento das implementações [SEL93a]. Também foi criado um aplicativo mais robusto para checar a estrutura dos diretórios e dos dados, semelhante ao *fsck* do FFS.

Serão apresentadas a seguir as modificações mais profundas e que mais influenciaram o desenvolvimento do nosso trabalho.

3.2.1 Organização física do sistema

Para aumentar a segurança do sistema, o BSD LFS replica o **superbloco** diversas vezes pelo disco. Desta forma, alguns segmentos possuem um superbloco no início, o que é indicado por um *flag* na tabela de utilização de segmentos. Não são mais necessárias regiões separadas de *checkpoint*, pois um arquivo especial chamado `ifile` (seção 3.2.3) guarda as informações sobre os metadados. O endereço do inode do `ifile` é guardado no superbloco e não há necessidade de replicar o `ifile`, uma vez que este pode ser reconstruído a partir dos sumários de segmento. A Figura 3.5 descreve a organização de um sistema de arquivos BSD LFS.

3.2.2 Segmentos

Existe uma diferença de notação entre as duas implementações que deve ser explicada: o BSD LFS trata a unidade formada por um sumário mais os blocos que este descreve pelo nome de **segmento parcial**, enquanto que o Sprite LFS trata esta mesma unidade por região de *log*.

No Sprite LFS, o sumário é sempre gravado no final do segmento (ou de uma região de *log*). Assim, espera-se que a aparição de um sumário indique que as informações às quais este se refere já tenham sido gravadas e, portanto, sejam consistentes. Infelizmente, nem sempre isso pode ser assegurado. Vários controladores de disco modernos possuem *buffers* internos que guardam uma trilha inteira de dados e podem iniciar a sua gravação em qualquer ponto do disco, não necessariamente no primeiro bloco requisitado. Isso é feito para diminuir o tempo de latência rotacional. Deste modo, pode ocorrer uma falha logo após o sumário ser gravado, sem que os dados aos quais este se refere estejam salvos.

Para evitar o problema acima, o BSD LFS implementa um *checksum* no sumário, formado por quatro bytes de cada bloco de dados do segmento parcial. O sumário também é salvo antes dos dados e os segmentos parciais são colocados na ordem lógica dentro do segmento e não na ordem inversa, como ocorria no Sprite LFS. Isso permite a gravação de vários segmentos parciais sem que seja necessária uma rotação completa do disco. A Figura 3.6 traz a diferença do arranjo dos segmentos.

Outra modificação no sumário é que as informações sobre os blocos que compõem o segmento estão separadas em estruturas, que correspondem aos arquivos aos quais estes pertencem. Assim, o sumário é composto por estruturas chamadas `FINFO` que contêm o número do inode, sua versão, o número de blocos do arquivo que estão no segmento e, para cada bloco, o endereço lógico e físico. Como não existem blocos de *log* de operações de diretório (seção 3.2.4), os únicos blocos, além dos sumários, que não estão em nenhuma estrutura `FINFO` são os blocos de inodes.

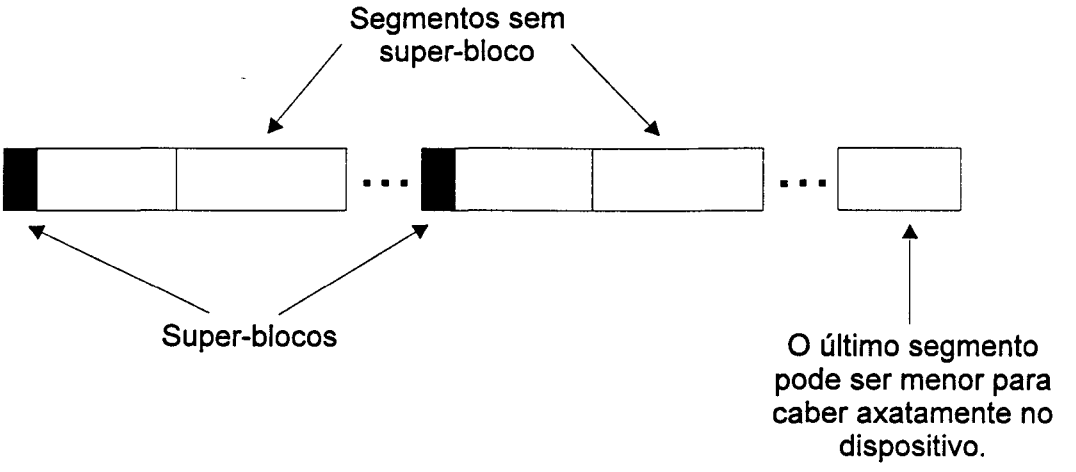


Figura 3.5: Organização física do BSD LFS.

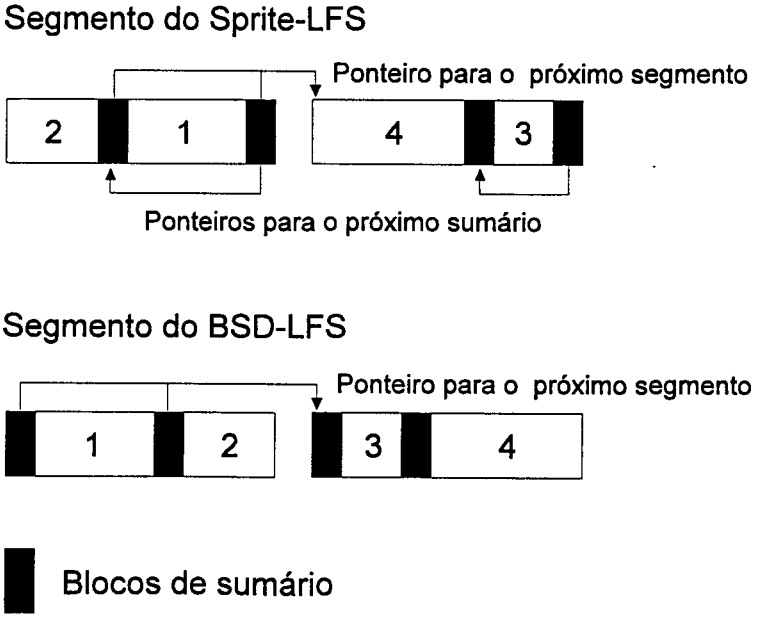


Figura 3.6: Organização do segmento no Sprite LFS e no BSD LFS.

3.2.3 O ifile

A modificação mais profunda do BSD LFS é quanto às estruturas especiais do LFS. Ao contrário do Sprite LFS, não existem mais os vetores de blocos do mapa de inodes (seção 3.1.2) e da tabela de utilização de segmentos (seção 3.1.4). Para substituir estas duas estruturas e armazenar o mapa de inodes e a tabela de utilização de segmentos foi criado um arquivo especial denominado `ifile`, que é visível por processos de usuários com permissão para leitura. O `ifile` é dividido em três partes: um cabeçalho, onde estão informações que o *cleaner* utiliza, a tabela de utilização de segmentos, que possui tamanho fixo, e o mapa de inodes, que cresce segundo a necessidade de novos inodes. Cada entrada da tabela de utilização de segmentos é representada por uma estrutura chamada `SEGUSE` e cada entrada do mapa de inodes é representada por uma estrutura chamada `IFILE`. A organização geral do `ifile` é mostrada na Figura 3.7.

A primeira vantagem que o `ifile` oferece é que o tamanho do *checkpoint* diminui, reduzindo-o à algumas dezenas de bytes e possibilitando que este fique no mesmo bloco de dados que o superbloco.

No Sprite LFS o número de inodes no sistema era limitado pelo número de blocos que o mapa de inodes continha. Para não aumentar demais o tamanho da tabela de blocos, que fica residente o tempo todo em memória, o número de inodes no sistema era calculado levando-se em conta que a média do tamanho dos arquivos seria 2KB. Embora esta estimativa tenha demonstrado ser suficiente, o sistema corria o risco de ficar sem inodes para criar novos arquivos. Com o `ifile`, cada vez que se necessita de mais inodes, um novo bloco de dados do arquivo é alocado. Desta forma, na prática, o total de inodes que podem ser criados é ilimitado.

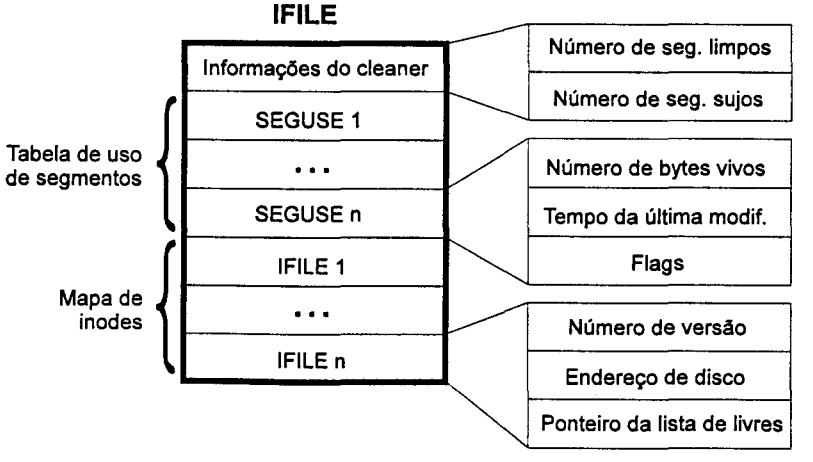


Figura 3.7: O ifile.

O `ifile` ainda guarda uma lista encadeada de inodes livres. Quando um novo arquivo é criado, o sistema verifica esta lista primeiro para ver se existe algum inode já alocado e que não está sendo utilizado no momento. A cabeça desta lista está na região de *checkpoint*. Assim, a procura por inodes livres no BSD LFS é mais simples e rápida. No Sprite LFS, o sistema tinha que procurar seqüencialmente pelo mapa de inodes até encontrar um inode vago, o que poderia levar algum tempo, caso fosse necessário acessar blocos do mapa que não estivessem na *cache*.

Por último, o fato do `ifile` ser visível por processos a nível de usuário permite o desenvolvimento de *cleaners* implementados fora do núcleo do sistema (seção 3.2.5), que foi uma das melhores inovações do BSD LFS.

Infelizmente, devido à características do UNIX BSD, o tempo de acesso do arquivo foi mantido no inode e não no mapa de inodes, como faz o Sprite LFS. Esta é uma péssima estratégia pois cada acesso ao inode, mesmo para leitura, torna-o "sujo" e força sua gravação. Isso contribui para o esgotamento do espaço livre do disco e, pior, faz com que o inode fique mais longe dos seus blocos de dados, anulando a localidade espacial característica dos segmentos.

3.2.4 Operações de diretório

Enquanto o Sprite LFS utiliza um *log* de operações de diretório para manter a consistência da estrutura de diretórios após alguma falha do sistema, o BSD LFS utiliza-se de uma técnica batizada de *segment batching*.

Quando uma operação de diretório é realizada, todos os inodes que fazem parte desta são marcados. Ao montar o segmento parcial para ser gravado, o sistema verifica se existem inodes marcados para serem gravados e, se houver, marca o segmento como contendo o início de uma operação de diretório. Os blocos de dados sujos do `ifile` e seu inode são salvos neste segmento, formando um *checkpoint*. Os próximos segmentos parciais também são marcados, até que todos os inodes participantes de operações de diretório sejam gravados. Normalmente, todos os inodes marcados cabem no mesmo segmento parcial.

Durante a recuperação do sistema, o aplicativo verifica o último *checkpoint* e realiza o processo de *roll-forward*. Se for encontrado algum segmento que inicia uma operação de diretório, qualquer operação posterior, já que as operações que incluem diretórios não podem ser identificadas, só serão levadas em conta se o segmento parcial que marcar o fim da operação de diretório tiver sido gravado corretamente.

Embora simplifique o *log*, eliminando uma estrutura redundante, esta alternativa para as operações de diretório provoca o aumento do número de gravação de *checkpoints* e também pode provocar a perda de diversas operações que não incluem diretórios e que poderiam ser totalmente restabelecidas depois de uma falha. O sistema pode perder estas operações exatamente porque, após encontrar uma marca de início de operações de diretórios, os segmentos parciais posteriores são totalmente descartados, caso não seja encontrada a marca de fim de operações de diretório.

3.3 Outros trabalhos

Diversos outros trabalhos, além dos mencionados acima, foram e estão sendo realizados sobre o LFS. Isso comprova o interesse neste modelo de sistema de arquivos, cujo potencial ainda não foi totalmente estudado. Vamos agora apresentar alguns dos mais interessantes e promissores trabalhos realizados.

3.3.1 Digital Spiralog

Uma implementação comercial do LFS é o Spiralog [DEC96, JJE96], feita pela Digital para o sistema operacional OpenVMS Alpha. O sistema aproveita-se das características do novo sistema de arquivos para aumentar o desempenho de seu modelo cliente-servidor, no qual cada cliente possui uma *cache* de dados *write-behind*. Outra vantagem é a velocidade com que são feitos *backups* incrementais, pois os dados novos estão sempre no final do *log* [GRE96]. Assim, o *log* é copiado bloco a bloco, sem nenhum processamento, desde o sumário imediatamente anterior ao horário escolhido até o último *checkpoint*. Caso o sistema de arquivos necessite ser restabelecido, devido à perdas de conteúdo de alguma região do *log*, um mecanismo similar ao *roll-forward* é utilizado para reconstrui-lo a partir dos *backups*.

O Spiralog também é interessante pois demostra que outros modelos de sistemas de arquivos podem utilizar o *log* da mesma forma que o LFS. O Spiralog não é baseado no sistema de arquivos do UNIX e por isso não implementa a sua semântica. Desta forma, os arquivos não são representados por inodes, mas por *headers*. A organização física dos arquivos também é diferente. Cada arquivo é dividido em extensões, indexadas por uma árvore B. Os diretórios também são indexados desta forma. Estas novas estruturas são gravadas no *log*, juntamente com os dados dos arquivos [WHI96].

3.3.2 Compressão de dados

Pesquisas da Digital também foram efetuadas sobre compressão de dados utilizando-se LFS [BUR92]. Os autores verificaram que o LFS é mais indicado para a implementação de compressão que sistemas de arquivos convencionais, pois escreve um montante grande de dados no disco de uma só vez. Isto favorece o trabalho de algoritmos adaptativos, que reconhecem padrões nos dados em uma região do bloco e utilizam esta informação para aumentar a compactação do restante do bloco.

Um problema com a compressão de dados é que blocos modificados podem ser rescritos com taxas de compactação que diferem daquelas dos blocos originais. Se a taxa for maior que a original, ao rescrever-se os dados, existe uma perda do espaço não utilizado no bloco. Se a taxa for menor, os dados não cabem mais no mesmo bloco e devem ser escritos

em outro, o que pode prejudicar a localidade dos blocos de um arquivo. Os dados no LFS não reutilizam a mesma região quando são modificados e rescritos, o que economiza espaço e diminui a fragmentação do disco. Além disso, o LFS não é tão dependente da localidade espacial dos dados quanto os sistemas convencionais.

Nos testes realizados em um protótipo, o desempenho do sistema sofreu uma pequena perda para utilização normal mas, em uso intensivo (como cópia de vários arquivos grandes), a perda chegou a 60%. Segundo os autores, este resultado é devido à utilização de compactação por *software*. Se fossem utilizados dispositivos de *hardware*, a perda em velocidade seria mínima ou irrelevante.

3.3.3 Zebra

Apesar do LFS não ser um sistema de arquivos intrinsecamente distribuído, sua utilização em sistemas operacionais distribuídos também é estudada. Nesta direção, foi desenvolvido um sistema que integra as tecnologias de LFS e RAID para conseguir um *throughput* 4 a 5 vezes maior que outros sistemas (Sprite e NFS) e tolerância parcial a falhas de servidores. Este sistema é chamado Zebra [HAR95].

O sistema Zebra utiliza o conceito de *stripe*, vindo da tecnologia RAID (seção 2.3.4), para dividir os arquivos entre diversos servidores distribuídos. Assim, quando um arquivo é requisitado, vários servidores podem ser utilizados em paralelo, diminuindo o tráfego a um servidor em especial e aumentando o número de operações de E/S que podem ser realizadas em um mesmo período de tempo.

As estações clientes no sistema Zebra utilizam *caches* para servir às requisições de leitura e agrupam operações de escrita da mesma forma que é feito no Sprite LFS. Estas requisições são enviadas aos servidores, que cuidam de gravá-las no *log*. Cada servidor pode conter unidades individuais de armazenamento ou um RAID. No caso de utilizar RAID, os servidores podem escolher subdividir os arquivos em *stripes* menores.

Os servidores utilizam informações redundantes que podem ser usadas numa eventual falha de um dos servidores. Assim, o sistema fica tolerante à falha de até um servidor de arquivos. Os clientes utilizam entre si um protocolo especial para manter a consistência dos dados armazenados em *cache*.

3.4 Análise do LFS

Alguns trabalhos [SEL93a, SEL95] podem levar a entender que os benefícios trazidos pelo LFS não compensariam trocar um sistema baseado no FFS, que é um modelo robusto e com ótimas implementações, por essa nova filosofia de sistema de arquivos. Estes trabalhos trazem comparações de desempenho entre implementações do LFS e do FFS e utilizam

programas de *benchmark* para coletar suas medidas. Na verdade, na maior parte das vezes estas medidas indicam a mesma coisa: o LFS é susceptível ao *workload* do sistema e possui baixo desempenho na leitura e alteração aleatórias de arquivos grandes. Porém, virtualmente nenhum sistema de arquivos está a salvo disso. Invertendo-se o ponto de referência, poderíamos dizer que o FFS possui desempenho até dez vezes menor que o LFS em termos de operações dominadas por metadados.

Além disso, o uso de *log* possui diversas outras qualidade, as quais serão explicadas abaixo. Também serão colocadas algumas características ruins do LFS e que ainda necessitam de estudos para minimizar o seu impacto sobre o desempenho do sistema.

3.4.1 Pontos fortes do LFS

O LFS é um sistema que alia um desempenho expressivo, especialmente no *workload* de ambientes de escritório, engenharia e ensino, com um alto potencial de aumento de desempenho. Por ser largamente baseado no uso de processador, uma vez que os algoritmos de ordenação dos segmentos e das estruturas auxiliares demandam um bom tempo de processamento, o LFS deve acompanhar o aumento de desempenho deste componente. Isso geralmente não ocorre com outros sistemas de arquivos, nos quais o uso do processador é menos intenso [ROS92b].

A tecnologia RAID já é uma realidade para servidores de arquivos e estações de trabalho para fins especiais, como computação gráfica. Porém, os RAIDs que possuem melhor custo/benefício, os de nível 4 e 5, apresentam um problema na gravação de pequenos arquivos (seção 2.3.4). Felizmente, o LFS pode agrupar diversas operações de gravação pequenas e enviá-las ao disco de uma só vez, eliminando este problema.

O processo de recuperação do LFS é muito mais simples que o dos outros sistemas de arquivos baseados no FFS. Isso acontece porque o FFS atualiza as informações no mesmo local onde estavam gravadas. Assim, quando ocorrer uma falha, a cabeça de leitura/gravação pode estar em qualquer lugar no disco e o programa de recuperação deverá fazer uma análise de todo o sistema de arquivos para averiguar eventuais inconsistências. No LFS, a cabeça do disco está sempre posicionada no final do *log*, que pode ser encontrado seguindo-se as estruturas de índice, ou em uma das regiões de *checkpoint*.

Agregando-se algumas informações adicionais no sumário de segmento, pode ser implementado um mecanismo de versões de arquivos no LFS. Como os blocos de dados e os metadados antigos dos arquivos não são perdidos quando uma atualização é feita, basta procurar os metadados que correspondem à versão desejada, o que pode ser indicado pela hora de última alteração do inode. A única precaução a ser tomada é que estes dados e metadados, que podem ser marcados no sumário de segmento, não poderão ser removidos do *log* no processo de limpeza. Outro mecanismo interessante que pode ser implementado é um processo de "*undelete*", já que o inode e os blocos de dados dos arquivos permanecem inalterados, mesmo depois da remoção dos mesmos.

3.4.2 Pontos fracos do LFS

Vários pontos do projeto LFS foram melhorados desde a sua primeira implementação no sistema operacional Sprite. Porém, algumas características ainda necessitam ser aprimoradas.

As estruturas auxiliares do LFS, como o mapa de inodes e a tabela de utilização de segmentos, permanecem em *cache* para aumentar seu desempenho. Por isso, o sistema necessita de um processo de *checkpoint*, realizado em períodos fixos de tempo, para assegurar a consistência em caso de falha. Quanto menor o intervalo de tempo entre *checkpoints*, menor será o desempenho do sistema. Por outro lado, quanto maior o intervalo, maior o tempo gasto para recuperar o sistema e maior a probabilidade de perda de informações.

A eficiência do sistema depende do *workload* ao qual este é submetido. Em sistemas que possuem uma taxa de utilização muito alta do disco (acima de 70%) o *cleaner* pode degradar demasiadamente o desempenho. Isso ocorre principalmente em instalações onde existem poucos períodos de ociosidade do sistema de arquivos, como por exemplo, transações (bancos de dados que atualizam poucos bytes em registros e forçam `fsync`) e atualizações aleatórias em arquivos.

Finalmente, as implementações feitas até agora ainda não estão maduras. Para sistemas críticos ou sistemas multi-usuários, a perda de dados devido a problemas nas implementações não pode ser tolerada. Assim, ainda existe uma certa resistência na adoção do LFS como sistema de arquivos principal nos sistemas UNIX.

Capítulo 4

O ambiente operacional LINUX

O LINUX foi escolhido como ambiente operacional para esta implementação do LFS por várias razões. Em primeiro lugar, este é compatível com o UNIX, o que faz seu sistema de arquivos nativo, o Ext2, possuir a mesma semântica do LFS. Este possui uma camada VFS bem implementada e que permite a adição de novos tipos de sistemas de arquivos, inclusive quando o sistema operacional já está em operação. Os códigos dos sistemas de arquivos suportados pelo LINUX estão disponíveis para pesquisa, o que facilita muito a implementação de novos sistemas de arquivos.

Serão apresentadas agora algumas características interessantes do LINUX e que mostram a qualidade deste sistema. Algumas funcionalidades como módulos, *buffer cache* e VFS serão descritas com maior profundidade, pois são essenciais ao nosso trabalho. Depois, será mostrado o sistema de arquivos Ext2, que é o mais utilizado no LINUX atualmente. O capítulo termina com uma explicação sobre a ferramenta UserFS, desenvolvida para o LINUX, que utilizamos em nosso trabalho.

4.1 O LINUX

O LINUX é um sistema operacional baseado no UNIX e desenvolvido por uma série de colaboradores espalhados pelo mundo. O idealizador do projeto foi Linus Torvald, responsável direto por grande parte de sua implementação original. A implementação foi realizada a partir do zero, mas sofreu influências de diversos sistemas como o MINIX e o FreeBSD. Ainda hoje a inclusão de novos códigos no LINUX é controlada por Torvald.

Inicialmente criado para ser executado em PCs 386, o LINUX já foi portado para diversas arquiteturas modernas, dentre as quais o MIPS, Alpha, SPARC e Pentium Pro. Atualmente, o núcleo do LINUX suporta até mesmo arquiteturas multi-processadas. O LINUX é compatível com o padrão POSIX, mas agrega a este modelo diversas características de outros padrões, tais como o BSD e o System V.

Várias características de sistemas operacionais modernos tais como VMS (*Virtual Memory System*), VFS, módulos e IPC, foram adicionadas ao sistema. Ao mesmo tempo, as mais diversas ferramentas, em especial as do projeto GNU, foram portadas para o LINUX. Portar programas para o LINUX geralmente é muito simples, uma vez que este possui uma interface bem definida e compatível com os principais padrões de UNIX atuais.

Por ser desenvolvido por diversas pessoas com objetivos diferentes, livre de diretrizes de empresas ou outras organizações, o núcleo do LINUX tem a vantagem de ser muito flexível e adaptável. Novas capacidades são adicionadas constantemente, o que pode ser observado pela quantidade e variedade de dispositivos de *hardware* suportados pelo núcleo. Estes incluem adaptadores de vídeo e de rede, sistemas de gerência de energia, discos rígidos (incluindo RAIDs), acionadores de CDROM, placas de som e fax-modems, dentre muitos outros.

Como o LINUX é um sistema livre, cujos códigos do núcleo são fornecidos com o pacote do ambiente, o usuário pode gerar um núcleo personalizado segundo suas necessidades. As opções que podem ser escolhidas no momento da compilação do núcleo são inúmeras e incluem suporte a diferentes configurações de *hardware* e a diversas funcionalidades, tais como protocolos de rede (TCP/IP, IPX, PPP, Appletalk), funções de *gateway* e *firewall*, IPC (no estilo do System V), quotas de disco para usuários e diversos tipos de sistemas de arquivos.

Embora seja um *software* inteiramente gratuito, várias empresas atualmente distribuem o LINUX em pacotes. Estes diferem um pouco entre si, em especial nas ferramentas de mais alto nível que são agregadas. Estas empresas cobram por serviços como a confecção de manuais de instalação, suporte ao usuário e algumas ferramentas não gratuitas. As distribuições mais conhecidas são a Slackware (que estamos utilizando), Redhat e Debian. Geralmente estas distribuições podem ser adquiridas gratuitamente através de servidores de arquivos na Internet.

A documentação sobre LINUX também é vasta. Esforços como o LINUX Documentation Project estão sendo realizados para melhorar mais a qualidade das informações sobre este sistema. Vários livros que abordam diversos aspectos deste sistema operacional também podem ser encontrados. Diversas publicações periódicas trazem artigos sobre o LINUX e pelo menos uma, The LINUX Gazette, dedica-se exclusivamente às notícias sobre o assunto.

4.1.1 Módulos

Módulos são um recurso utilizado pelo LINUX e outros sistemas operacionais modernos baseados no UNIX para dar um pouco mais de flexibilidade ao seu núcleo monolítico [JMK95, VAH96]. Um módulo é uma biblioteca binária que é ligada ao núcleo do sistema operacional após este ter sido carregado e inicializado. Um módulo pode ser carregado automaticamente quando necessário ou por intervenção direta do administrador do

sistema. Além disso, um módulo pode ser retirado do núcleo a qualquer momento, desde que não esteja sendo utilizado.

Um módulo geralmente possui código bastante especializado para estender alguma funcionalidade do núcleo ou para manipular certos dispositivos. Vários *drivers* de dispositivos, como CDROM, discos removíveis e dispositivos *multimedia*, quando são pouco utilizados no sistema, podem ser implementados como módulos e ser carregados apenas quando forem necessários.

As duas grandes vantagens dos módulos sobre o código estático é que seu uso diminui o tamanho do núcleo e simplifica seu código. A primeira característica libera mais memória para ser utilizada por aplicações ordinárias. A segunda torna o núcleo potencialmente mais rápido, uma vez que menos símbolos devem ser verificados nas chamadas de sistema.

Uma vez que um módulo é carregado, este se cadastra e passa a fazer parte do núcleo do sistema operacional. Assim, os módulos não estão sujeitos às limitações aplicadas aos processos de usuários, como mudança do espaço de endereçamento ou escalonamento. Isto permite que os módulos sejam executados sem nenhum *overhead*. Porém, se o seu código contiver erros, o sistema poderá entrar em um estado inconsistente e ficar paralisado. Desta forma, é necessário cautela ao se desenvolver ou utilizar módulos. Por isso, apenas o administrador do sistema possui privilégios para carregá-los e descarregá-los.

Nas versões do LINUX a partir da 2.0.0, várias funcionalidades do núcleo podem ser compiladas como módulos, incluindo o suporte a sistemas de arquivos, som, *mouse* e dispositivos de armazenamento, entre outros. O ambiente conta ainda com um processo *daemon* chamado **`kerneld`**, capaz de controlar a carga e descarga de módulos. Quando o sistema é inicializado, uma função interna do núcleo chama `kerneld`, que verifica em um arquivo especial do sistema, chamado `/etc/conf.modules`, quais os módulos que devem ser carregados para a memória. O processo `kerneld` é capaz de controlar o acesso a estes módulos e descarregar alguns deles que não estejam sendo utilizados. Tal decisão pode ser tomada para liberar espaço na memória principal para as aplicações ou para o próprio núcleo. O sistema ainda guarda no arquivo `/etc/conf.modules` os nomes de outros módulos que podem ser carregados caso seja feita alguma referência a um símbolo definido nestes.

Uma utilização interessante dos módulos é permitir que o núcleo do sistema suporte inúmeros tipos de sistemas de arquivos. Por exemplo, o suporte ao sistema de arquivos nativo do LINUX, o Ext2, e ao NFS podem ser compilados e ligados em tempo de criação do núcleo do sistema, por serem muito utilizados e essenciais. Por outro lado, o suporte a sistemas menos utilizados como o MinixFS e o DOS-FAT podem ser compilados como módulos e carregados quando alguma partição ou disco de algum destes modelos forem montados. Estes módulos podem ser descarregados assim que o sistema de arquivos for desmontado. Desta forma, diversos tipos de discos e partições podem ser entendidos pelo núcleo sem que este possua um tamanho ou uma complexidade proibitivos para computadores com menor capacidade. A implementação de sistemas de arquivos como módulos só é possível porque o LINUX conta com uma camada VFS totalmente dinâmica.

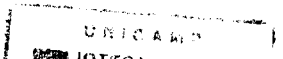
4.1.2 Buffer cache

Assim como virtualmente todos os sistemas operacionais modernos, o LINUX possui um sistema de gerenciamento de memória virtual, ou VMS, integrado à memória principal e ao subsistema de arquivos do sistema operacional. No caso do LINUX, o subsistema de arquivos é representado pela camada VFS e pelas implementações dos tipos suportados. A ponte de ligação entre o VMS e o VFS do LINUX é a *buffer cache* [WIR95]. Esta *cache* de blocos de disco é peça chave para os sistemas de arquivos que utilizam dispositivos de armazenamento, pois todas as requisições de leitura e escrita de blocos passam obrigatoriamente pela *buffer cache* antes de chegarem aos *drivers* dos dispositivos. Em sistemas que não necessitam de dispositivos de blocos, tais como o NFS e o UserFS, as chamadas são direcionadas para outros meios, como os *drivers* de rede, no caso do NFS, ou o processo de usuário, no caso do UserFS.

O objetivo principal da *buffer cache* é aumentar o desempenho dos sistemas de arquivos, eliminando diversos acessos ao disco (seção 2.3.3). Além disso, no LINUX, esta *cache* conta com algoritmos de leitura antecipada (*read ahead*) e escalonamento de gravações de blocos. O espaço em memória reservado para a *cache* é variável em tempo de execução e depende da carga do sistema. Quando o núcleo é inicializado, o sistema reserva toda a memória virtual livre para a *buffer cache*. Conforme novos processos vão sendo criados e exigem novas páginas de código e dados do VMS, o tamanho da *cache* é diminuído até um valor mínimo estipulado no núcleo. Se algumas páginas são liberadas pelos processos, estas retornam automaticamente para a *cache*. Caso esteja cheia, a *cache* é responsável por salvar blocos sujos de forma ordenada, mantendo a semântica das operações do sistema de arquivo, antes de liberar espaço para o VMS.

As estruturas principais da *buffer cache*, como o nome já diz, são os *buffers*. Cada *buffer* possui um cabeçalho com informações necessárias para sua manipulação e um apontador para uma área de dados que representa um bloco de um sistema de arquivos. As informações adicionais do cabeçalho incluem seu estado (sujo, bloqueado, protegido, acessado, etc.), o tempo de último acesso ao bloco – usado para implementar o algoritmo *Last Recently Used*, responsável por gravar blocos sujos no disco – e o tamanho do bloco de dados (que pode variar conforme o sistema de arquivos).

Os *buffers* são indexados pelo número do dispositivo do qual vieram e pela sua posição dentro do dispositivo. Esta abordagem difere de outros sistemas, como o BSD 4.4, nos quais os blocos de dados são indexados na *cache* pelo número do inode ao qual pertencem e pela sua posição lógica dentro do inode. Esta forma de indexar blocos usada no LINUX pode penalizar certas operações que envolvem diversos blocos de um arquivo. No BSD, por exemplo, cada estrutura `vnode` em memória possui um apontador para a lista dos seus blocos de dados que estão em memória. Embora esta lista não seja ordenada pela posição lógica do bloco, é simples encontrar todos os blocos de um `vnode`, como é necessário no caso de uma operação `fsync` ou `truncate`. Já no LINUX, os endereços dos blocos são acessados a partir dos apontadores do `inode` (nome da estruturas na *cache*) e a pesquisa é feita na tabela de *hashing* de *buffers*, que é global para toda a *cache*. Dependendo da carga da *cache*, esta



procura pode ser um pouco mais demorada que no caso do BSD. Por outro lado, se uma chamada `sync`, que força a gravação de todos os blocos sujos existentes na *cache* (inclusive superblocos e inodes) for efetuada, o LINUX leva vantagem sobre o BSD, pois pode organizar melhor e mais rapidamente a ordem de gravação dos blocos de dados em disco.

Em sistemas UNIX em geral, os dados modificados permanecem em *cache* por 30s antes de serem salvos em disco (seção 2.3.3). O LINUX possui um esquema diferente de gravação. Este utiliza um *daemon*, chamado `bdflush`, que salva os *buffers* sujos, isto é, recentemente criados ou modificados, de maneira menos regular. O *daemon* `bdflush` tenta disseminar a gravação dos blocos durante os 30s para evitar o acúmulo de operações de disco a cada 30s. Este comportamento é útil, por exemplo, quando vários *buffers* sujos são gerados em um curto espaço de tempo.

O LINUX conta também com a chamada de sistema `mmap` em sua interface [DUB94a]. Quando um processo invoca `mmap` sobre um arquivo, os blocos mapeados não são postos na *buffer cache* e sim em páginas no espaço lógico de endereçamento do processo que invocou `mmap`. Assim, estes blocos não ficam sujeitos às políticas de gravação implementadas para os *buffers*, podendo permanecer em memória principal durante o tempo que o processo quiser. Eventualmente, estes blocos podem ser mandados para a área de *swap*, caso o VMS exija as páginas ocupadas por estes. Isso, porém, é feito de forma totalmente transparente ao usuário.

4.1.3 VFS

O VFS do LINUX segue as mesma idéias do VFS original da Sun [KLE86, SAN87], mas difere deste em certos detalhes de implementação. Isso, porém, não altera a interface do sistema de arquivos como um todo, que continua com as mesmas chamadas de sistemas padrão do UNIX. A utilização de um padrão de interface para o subsistema de arquivos facilitaria o processo de portar outros sistemas de arquivos para o LINUX. Porém, não existe um padrão amplamente aceito no mundo UNIX e o que pode ser encontrado são propostas que não trazem grandes vantagens sobre a interface do LINUX [KAR86, VAH96].

A camada VFS do LINUX é mais flexível que o modelo original e permite que novos tipos de sistemas de arquivos sejam agregados ao núcleo do sistema através de módulos (seção 4.1.1). Atualmente, o LINUX suporta mais de uma dezena de tipos de sistemas de arquivos. Estes são sistemas vindos tanto do mundo UNIX, como o Ext2, o NFS e o s5fs, quanto de sistemas operacionais diferentes, como o DOS-FAT (MS-DOS e compatíveis), MinixFS (MINIX), HPFS (OS/2 *High Performance File System*). O suporte a outros sistemas, como o NTFS (*New Technology File System*, da Microsoft) e o UFS (UNIX *File System*, da Sun), também está sendo desenvolvido. Porém, o núcleo de uma certa instalação do LINUX não necessita suportar todos estes sistemas de arquivos. No momento da compilação do núcleo, o usuário pode escolher quais sistemas serão suportados. Também pode ser escolhido o suporte de alguns tipos por meio de módulos. Assim, o sistema encarrega-se de carregar automaticamente os módulos corretos quando houver necessidade (seção 4.1.1).

Como no VFS original (seção 2.3.5), os principais objetos que formam a camada VFS são os sistemas de arquivos montados, representados no LINUX por estruturas do tipo `vfsmount` e `super_block`, e os inodes, representados por estruturas `inode`. Note que o LINUX não utiliza a mesma terminologia do VFS original, onde os inodes mapeados em memória são chamados `vnodes`. Um outro objeto importante é o tipo de sistema de arquivos, representado por `file_system_type`. Para que seja possível acompanhar de forma mais clara a explicação da dinâmica da camada VFS, as suas principais estruturas estão sumariadas abaixo:

- **`file_system_type`**: corresponde a um tipo de sistema de arquivos suportado pelo núcleo. Guarda informações que serão usadas quando um sistema de arquivos é montado, dentre as quais, um apontador para uma função `read_super`, usada para ler o superbloco do sistema de arquivos, e o campo `name`, que representa o nome do tipo de sistema de arquivos que representa.
- **`vfsmount`**: representa cada sistema de arquivos montado. Seus principais campos são `mnt_dev`, que é o número do dispositivo onde o sistema se encontra (caso seja necessário), e `mnt_sb`, que é um apontador para o superbloco do sistema de arquivos. As estruturas `vfsmount` formam uma lista ligada.
- **`super_block`**: guarda as informações gerais sobre o sistema de arquivos necessárias ao VFS. Possui, em especial, um apontador chamado `s_op` para uma estrutura do tipo `super_operations`, que guarda as operações suportadas pelo sistema de arquivos. Possui também um campo, `u`, que guarda informações particulares exigidas pelas implementações de cada tipo de sistema de arquivos. A estrutura `super_block` ainda traz dois apontadores para estruturas `inode`, utilizados na operação `mount`. O primeiro, `s_covered`, indica o inode do diretório que é ponto de montagem do sistema de arquivos. O segundo, `s_mounted`, indica qual o inode raiz do sistema.
- **`inode`**: representa cada inode em memória. Possui, entre outros campos, um apontador chamado `i_op` para uma estrutura do tipo `inode_operations`, que guarda as operações específicas do tipo de sistema de arquivos suportadas pelos inodes. Possui também um apontador para `inode`, chamado `i_mount`, que é utilizado quando o inode é um ponto de montagem. O campo `i_mount` aponta para o inode raiz do sistema de arquivos montado.
- **`file`**: representa um arquivo aberto. Possui um apontador, `f_inode`, para o inode ao qual se refere e outro apontador, chamado `f_op`, para uma estrutura do tipo `file_operations`, que guarda as operações permitidas apenas sobre arquivos abertos. As operações permitidas sobre arquivos incluem `lseek`, `read`, `write`, `readdir`, `ioctl` e `fsync`, entre outras.

As interfaces para os tipos principais do VFS, `super_block`, `inode` e `file`, estão na Figura 4.1. Várias das operações apresentadas são necessárias às implementações tanto do VFS como do VMS do LINUX.

```
struct super_operations {
       void (*read_inode) (struct inode *);
       int (*notify_change) (struct inode *, struct iattr);
       void (*write_inode) (struct inode *);
       void (*put_inode) (struct inode *);
       void (*put_super) (struct super_block *);
       void (*write_super) (struct super_block *);
       void (*statfs) (struct super_block *, struct stafs *, int);
       int (*remount_fs) (struct super_block *, int *, char *);
};


struct inode_operations {
       struct file_operations *default_file_ops;
       int (*create) (struct inode *, const char *, int, int, struct inode**);
       int (*lookup) (struct inode *, const char *, in, struct inode **);
       int (*link) (struct inode *, struct inode *, const char *, int);
       int (*unlink) (struct inode *, const char *, int);
       int (*symlink) (struct inode *, const char *, int, const char *);
       int (*mkdir) (struct inode *, const char *, int, int);
       int (*rmdir) (struct inode *, const char *, int);
       int (*mknod) (struct inode *, const char *, int, int, int);
       int (*rename) (struct inode *, const char *, int, struct inode *,
            cost    char *, int);
       int (*readlink) (struct inode *, char *, int);
       int (*follow_link) (struct inode *, struct inode *, int, int,
            struct inode *);
       int (*readpage) (struct inode *, struct page *);
       int (*writepage) (struct inode *, struct page *);
       int (*bmap) (struct inode *, int);
       void (*truncate) (struct inode *);
       int (*permission) (struct inode *, int);
       int (*smap) (struct inode *, int);
};


struct file_operations {
       int (*lseek) (struct inode *, struct file *, off_t, int);
       int (*read) (struct inode *, struct file *, char *, int);
       int (*write) (struct inode *, struct file *, const char *, int);
       int (*readdir) (struct inode *, struct file *, void *, filldir_t);
       int (*select) (struct inode *, struct file *, int, select_table *);
       int (*ioctl) (struct inode *, struct file *, unsigned int,
            unsigned long);
       int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
       int (*open) (struct inode *, struct file *);
       void (*release) (struct inode *, struct file *);
       int (*fsync) (struct inode *, struct file *);
       int (*fasync) (struct inode *, struct file *, int);
       int (*check_media_change) (kdev_t dev);
       int (*revalidade) (kdev_t dev);
};
```

Figura 4.1: Interfaces dos principais objetos do VFS do LINUX.

Em `super_block`, por exemplo, temos `put_inode` e `put_super`, utilizadas para informar a *buffer cache* que, respectivamente, uma estrutura `inode` ou `super_block` não é mais necessária e pode ser liberada, ou seja, destruída. É bom lembrar, no entanto, que estas

interfaces serão vistas apenas internamente ao núcleo do sistema, pelas implementações das chamadas de sistemas do LINUX. Assim, nem mesmo as bibliotecas de E/S, como `libc`, saberão destes detalhes de implementação.

Os tipos de sistemas de arquivos suportados pelo núcleo do LINUX são guardados em uma lista encadeada e não em uma tabela de tamanho fixo, como ocorria em algumas implementações mais antigas do VFS. Assim, novos tipos de sistemas de arquivos podem ser agregados apenas inserindo uma nova estrutura do tipo `file_system_type` nesta lista. As estruturas são inseridas e removidas na lista de sistemas de arquivos suportados pelas funções internas do sistema `register_filesystem` e `unregister_filesystem`. A função `register_filesystem` é chamada internamente no código do núcleo do sistema quando este está sendo carregado e inicializado, ou então por um módulo que implementa um sistema de arquivos, quando este módulo é carregado. A função `unregister_filesystem` é chamada quando um módulo é descarregado.

Quando uma chamada de sistema `mount` chega ao subsistema de arquivos, a primeira coisa que o núcleo faz é procurar a estrutura `file_system_type` referente ao tipo do sistema de arquivos especificado como parâmetro da chamada. Encontrando esse tipo, `mount` então invoca a função `read_super` indicada na estrutura. Os passos realizados por `read_super` dependem do tipo de sistema de arquivos em questão. Geralmente é criada uma estrutura `super_block`, que é inserida na lista de superblocos e preenchida com as informações lidas a partir do superbloco residente na partição. A lista de superblocos é uma forma simples do LINUX encontrar quais os sistemas de arquivos montados que estão associados a algum dispositivo. Esta informação é utilizada, por exemplo, quando uma chamada `sync` é realizada, forçando todos os dados de todos os sistemas de arquivos a serem gravados em disco. Outro passo importante é fazer `s_op` apontar para o código correto das operações do superbloco para o tipo em questão. Todos os sistemas de arquivos de um mesmo tipo que forem montados compartilharão o mesmo código das operações. Após inicializar o superbloco, `read_super` normalmente lê o inode raiz do sistema de arquivos, que é então associado ao apontador `s_mounted`.

Depois que o superbloco está preenchido, `mount` cria uma estrutura do tipo `vfsmount`, associa o apontador `mnt_sb` ao superbloco recentemente criado e a coloca na lista de sistema de arquivos montados. O VFS do LINUX difere do VFS da Sun pois separa a estrutura que representa o sistema de arquivos montado, `vfsmount`, da estrutura que representa o superbloco, `super_block`. Por último, o apontador `s_covered` do superbloco recebe o valor do inode que representa o ponto de montagem do novo sistema de arquivos e o apontador `i_mount` deste inode recebe o valor de `s_mounted` do superbloco, ou seja, passa a apontar para o inode raiz do sistema. No VFS original, o ponto de montagem possui um apontador para o objeto `vfs`. A razão pela qual o LINUX faz esta mudança parece ser por motivos de desempenho.

Quando um inode é criado ou lido, uma estrutura `inode` é criada em memória. Preencher os dados de `inode` também é tarefa da implementação do sistema de arquivos. Geralmente, um dos passos realizados é fazer o apontador `i_op` apontar para o conjunto correto de operações para aquele tipo de inode. Por exemplo, um inode que representa um dispositivo, ou outro arquivo especial qualquer, terá operações diferentes de um inode que

representa um arquivo comum. Assim, cada tipo de sistema de arquivos deve prover estruturas diferentes com as operações necessárias para cada tipo de inode. As operações suportadas podem ser implementadas pelo código específico do sistema de arquivos ou por funções padrão implementadas de forma global na camada VFS. Desta forma, por exemplo, quando forem criados dois novos arquivos de dispositivos (*device special files*), um em um sistema do tipo Ext2 e outro no UFS, ambos poderão usar o mesmo código global. Isso pode ser feito pois arquivos de dispositivos não dependem da organização física dos dados que os sistemas de arquivos possuem, uma vez que não armazenam realmente dados, mas são apenas um ponto de acesso aos *drivers* dos dispositivos.

O VFS do LINUX faz distinção entre arquivos que estão abertos, representados por um par de estruturas `file + inode`, e arquivos que não estão abertos, representados apenas por `inode`. Cada uma destas estruturas possui um conjunto diferente de operações que podem ser realizadas sobre estas, como pode ser visto na Figura 4.1. Esta modelagem está mais próxima da realidade, pois apenas arquivos abertos, dos quais o processo detém um descritor, podem, por exemplo, ser lidos, escritos ou mapeados em memória.

4.1.4 Second Extended File System

O primeiro sistema de arquivos suportado pelo LINUX foi o MINIX *File System* (MinixFS), originário do sistema operacional MINIX [TAN87]. Este sistema, porém, apresentava limitações, tais como tamanho máximo dos nomes de arquivos de 14 caracteres e tamanho máximo do sistema de arquivos de 64MB. Assim, um novo sistema de arquivos, chamado *Extended File System* (Extfs) foi desenvolvido para substituir o MinixFS.

Embora eliminasse as principais limitações do MinixFS, o Extfs possuía um desempenho ruim, devido a simplicidade de sua implementação [CAR93]. Assim, em 1993 foi liberada a primeira versão do LINUX com um novo sistema de arquivos, o *Second Extended File System*, ou Ext2fs [CAR93, DUB94b]. Este sistema é uma implementação do FFS [MCK84], influenciado especialmente pelo BSD FFS e agregado de algumas características interessantes, tais como leitura antecipada de blocos, semântica de arquivos estendida e *links* simbólicos pequenos embutidos nos inodes.

O Ext2 divide o disco, ou partição, em grupos de blocos (*block groups*), que correspondem aos grupos de cilindros do FFS (seção 2.4.2). Cada grupo de bloco possui, nesta ordem:

- Uma cópia redundante do superbloco do sistema de arquivos. Este é replicado pelo disco para segurança e possui informações gerais do sistema de arquivos.
- Um descritor do grupo (também chamado descritor do sistema de arquivos). Este possui informações locais a cada grupo. Estas incluem o número de inodes livres, o número de blocos livre e o número de inodes alocados para diretórios no grupo de blocos.

- Um mapa de bits para a alocação de blocos. Fundamentalmente, este mapa possui um bit para cada fragmento do grupo. Porém, o Ext2 atualmente não implementa fragmentos, embora as suas estruturas internas provenham suporte para estes. Assim, cada bit do mapa eqüivale a um bloco de dados.
- Um mapa de bits para a alocação de inodes.
- Uma tabela de inodes do grupo.
- Uma área reservada para os blocos de dados.

Como não implementa fragmentos, o tamanho padrão dos blocos de dados no Ext2 é 1KB. Assim, o sistema impede a fragmentação do disco devido a perda de espaço no último bloco do arquivos. Com um tamanho de bloco menor, o número de blocos por arquivo aumenta consideravelmente. Como outro efeito, o número de apontadores de blocos por bloco de dados também diminui. Isso implica no uso de um terceiro nível de apontadores para blocos, ou blocos triplamente indiretos, no inode.

Quando um novo bloco de um arquivo é criado, o Ext2 aloca previamente até 8 blocos de dados adjacentes, esperando que o arquivo cresça. Assim, os arquivos tendem a ser gravados em agrupamentos no disco, o que melhora o desempenho de escrita e de leitura. Se o sistema necessitar do espaço dos blocos alocados previamente, estes serão liberados.

Com o auxílio da *buffer cache*, o Ext2 realiza leitura antecipada de blocos (*read ahead*) tanto em arquivos comuns quanto em diretórios. A leitura antecipada de blocos é disparada sempre que o sistema detecta um padrão de leitura seqüencial no arquivo. Isso também é essencial ao desempenho do sistema, uma vez que o tamanho reduzido do bloco de dados determina uma maior quantidade de blocos lidos em diversas operações de E/S.

A alocação de inodes e blocos nos grupos de blocos segue o mesmo algoritmo do FFS. Os diretórios são alocados balanceando-se o número de inodes alocados nos grupos. Os inodes dos arquivos criados em um diretório são alocados no mesmo grupo que o inode do diretório. Os blocos de um arquivo são alocados no mesmo grupo em que o inode se encontra. Blocos de um mesmo arquivo são alocados em seqüência. Em discos com taxas de utilização altas, nem sempre o sistema é capaz de seguir estas diretrizes e os inodes e blocos de dados são alocados conforme a disponibilidade de espaço nos grupos de blocos.

O Ext2 possui diversas características especiais que normalmente não são encontradas em outros sistemas de arquivos:

- Opção de semântica do sistema de arquivos do BSD ou do System V, que pode ser escolhida na hora em que o sistema é montado.
- Os metadados podem ser salvos síncrona ou assincronamente, dependendo da opção escolhida quando o sistema é montado.
- Os *links* simbólicos até 60 caracteres são guardados no próprio inode, no local reservado para os apontadores de blocos. Isso aumenta o desempenho na leitura dos *links* e diminui a fragmentação do espaço do sistema de arquivos.

- Remoção segura de arquivos, indicada por um atributo especial do arquivo. Quando o arquivo é apagado, o sistema grava dados aleatórios em todos os blocos que estavam alocados para este.
- Dois novos tipos de arquivos, inspirados no BSD 4.4. Arquivos *append-only* não podem ser apagados nem renomeados e novos dados são sempre adicionados no final do arquivo. Podem ser utilizados para implementar *logs*. Arquivos imutáveis só podem ser lidos e são importantes para configurações que não podem ser alteradas jamais.

4.2 User File System

Por ser um sistema operacional baseado no UNIX, todas as funcionalidades do LINUX, incluindo o subsistema de arquivos, são implementadas em um único núcleo monolítico, protegido do usuário. Desenvolver novas funções para este sistema operacional significa ter que alterar o código do núcleo, recompilá-lo e reinicializar o sistema fazendo-o utilizar a nova versão do núcleo. Outra solução é utilizar módulos, que agregam novas funções ao núcleo. O uso de módulos, porém, não diminui o risco de falhas no sistema causadas por erros no código do novo módulo.

Para alterar o código de um sistema operacional, o desenvolvedor deve possuir muita experiência no sistema que está alterando e afinidade com sua estrutura e funcionamento internos, pois o código do núcleo geralmente é intrincado e em certos aspectos dependente de uma arquitetura em especial. Recompilar o núcleo, por exemplo, geralmente requer o conhecimento prévio de dispositivos de *hardware* existentes na estação alvo. Além disso, na ocorrência de falhas, o restabelecimento do sistema pode ser bastante trabalhoso e demorado, principalmente se algum dano for causado a arquivos importantes do sistema. Isto torna o desenvolvimento de sistemas de arquivos novos mais crítico ainda. Por fim, o código do núcleo não é alcançável a nível de usuário, não sendo possível rodar sobre esse os aplicativos comuns para depuração (*debuggers*), o que dificulta bastante a sua correção.

Para facilitar e acelerar o desenvolvimento do protótipo, decidimos utilizar uma ferramenta chamada *User File System* (UserFS) [FIT93], que permite a implementação de sistemas de arquivos experimentais como processos comuns de usuário. Esta ferramenta está disponível para cópia e uso e seu código é livre para ser estudado e alterado.

4.2.1 Modelo do User File System

O UserFS fornece um canal de comunicação entre processos a nível de usuário e a camada VFS do LINUX, de forma que os primeiros possam responder às mesmas chamadas

de sistema feitas a sistemas de arquivos internos ao núcleo. Assim, sistemas de arquivos experimentais podem ser desenvolvidos e testados fora do núcleo do sistema operacional. A comunicação entre o núcleo do UserFS, que encontra-se incorporado ao núcleo do LINUX, e o processo de usuário se dá através de um protocolo específico da ferramenta. A Figura 4.2 explica como o UserFS se liga ao núcleo do LINUX e ao processo do usuário.

Uma característica importante do UserFS é que este permite que os sistemas experimentais sejam montados, ou seja, feitos visíveis como parte da árvore de sistemas de arquivos, por usuários comuns, o que é proibido para sistemas de arquivos normais. Manipular partições e dispositivos como usuário comum praticamente anula o risco de danos em outras partições que já foram formatadas para outro tipo de sistema de arquivos.

O UserFS permite também que diversas aplicações possam se utilizar da abstração do sistema de arquivos para manipular vários tipos de estruturas que não precisam estar diretamente ligadas à unidades de armazenamento de massa. Por exemplo, um sistema de arquivos pode utilizar uma conexão FTP para fornecer às aplicações a impressão de que arquivos que na verdade encontram-se em servidores remotos estão em um diretório local.

O funcionamento do UserFS é parecido com o do NFS da Sun, na medida em que ambos "empacotam" as chamadas do subsistema de arquivos e as enviam para fora do núcleo do sistema operacional. A maior diferença é que o NFS envia as requisições via rede para um servidor NFS no núcleo de outra máquina, enquanto o UserFS as envia para um processo qualquer em nível de usuário. Os protocolos utilizados pelos dois sistemas também são diferentes. O protocolo do NFS é *stateless*, ou seja, o servidor não guarda informações sobre os seus clientes nem sobre as requisições realizadas por estes, e é construído sobre o RPC (*Remote Procedure Call*) e o XDR (*Extern Data Representation*), ambos da Sun. Por sua vez, o UserFS utiliza um protocolo *statefull* independente de máquina, desenvolvido especialmente para a ferramenta (seção 4.2.4) e a representação dos tipos de dados do protocolo utiliza uma linguagem própria de definição de dados. É bom ressaltar que, mesmo com princípios de funcionamento parecidos, os propósitos dos dois sistemas de arquivos são bem distintos. O NFS é um sistema de arquivos comercial e distribuído, cuja arquitetura inclui clientes e servidores codificados no núcleo do sistema operacional, enquanto o UserFS é uma ferramenta destinada ao desenvolvimento de sistemas de arquivos experimentais.

Uma característica importante do protocolo do UserFS é que os processos de usuário não necessitam estar preparados para responder todas as operações previstas no protocolo. Assim, quando um novo sistema de arquivos é montado, o núcleo do UserFS faz uma chamada `up_enquire` ao processo de usuário para cada uma das operações do protocolo. Se o processo for capaz de realizar uma operação, este responde positivamente ao *enquire*, caso contrário, este retorna um erro. Esta abordagem permite que sistemas que não aceitam todas as operações previstas no protocolo possam ser implementados também. Por exemplo, um sistema de arquivos que manipula uma mídia só de leitura não poderia implementar operações que modificassem os dados. Além disso, esta característica do protocolo também permite que as operações sejam agregadas progressivamente ao processo, facilitando a depuração de seu código.

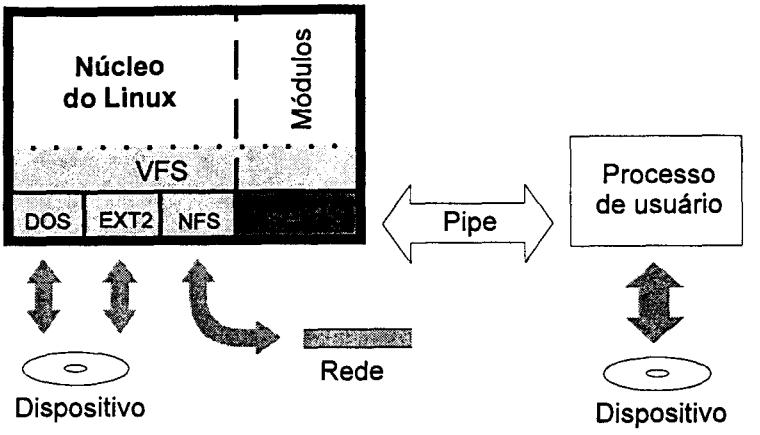


Figura 4.2: O UserFS ligado ao núcleo do LINUX e a um processo de usuário.

4.2.2 O núcleo

A principal peça do UserFS é seu núcleo, que é implementado como um módulo (seção 4.1.1), podendo ser carregado no núcleo do sistema operacional e agindo como mais um sistema de arquivos suportado por este. O núcleo, após instalado, tem capacidade de responder a vários processos de usuários ao mesmo tempo. Cada um destes processos implementa um sistema de arquivos experimental. O núcleo do UserFS não faz nenhuma suposição quanto a organização destes processos, nem quanto aos dados que estes manipulam. Este só faz duas exigências aos processos: a primeira é que possuam um descritor de arquivos por onde o núcleo possa enviar mensagens e outro descritor por onde possa receber mensagens; a segunda é que estes sejam capazes de responder às requisições feitas através do protocolo do UserFS. A única restrição sobre os descritores de arquivos que o núcleo faz é que um deles possua permissão de escrita e o outro de leitura. Assim, os descritores podem representar arquivos comuns, um *pipe* em memória ou *sockets*, o que permite que o processo de usuário esteja em uma máquina distinta da qual o núcleo se encontra. Porém, o mais comum é o processo encontrar-se na mesma máquina que o núcleo do UserFS e um *pipe* é geralmente utilizado para a comunicação.

Um dos principais objetivos buscados no desenvolvimento do núcleo do UserFS foi a sua robustez frente à falhas dos processo experimentais. Mesmo que um processo deixe de responder às requisições, o núcleo não deixa o sistema em um estado inconsistente ou de *deadlock*.

Quando o módulo que contém o núcleo do UserFS é carregado no núcleo do LINUX, esse se cadastra na camada VFS como o sistema de arquivos **`userfs`**. A partir daí, este passa a responder por qualquer requisição feita a um sistema de arquivos deste tipo. O núcleo é

responsável por manipular as estruturas `inode` em memória e sincronizar as operações sobre estes. O processo de usuário, por sua vez, é responsável por realizar as operações no dispositivo ou outro local onde os dados do sistema de arquivos se encontrem.

4.2.3 Operação mount

Um sistema de arquivos do tipo `userfs` é montado quase da mesma forma que um sistema de arquivos comum. Para o usuário, a diferença é que este deve utilizar a ferramenta **`muserfs`**, que integra o pacote UserFS, e não o comando `mount` do sistema operacional, que normalmente é utilizado. Os passos para realizar a montagem do sistema de arquivos estão descritos na Figura 4.3.

Caso o processo de usuário utilize um disco como armazenamento, o primeiro passo a ser realizado, como em outros sistemas comuns, é a formatação lógica da partição que conterá o sistema de arquivos. Embora esta operação não dependa do UserFS, este é mostrado na Figura 4.3a.

A ferramenta `muserfs` toma como parâmetro o nome do processo de usuário, o ponto de montagem do novo sistema de arquivos e uma série de parâmetros adicionais, dependentes do tipo de sistema de arquivos, que serão passados ao processo de usuário. Inicialmente, `muserfs` cria um *pipe* e dispara o processo de usuário, que herda o *pipe*, o qual utilizará para receber e responder às requisições do núcleo. Em seguida, `muserfs` invoca a chamada de sistema `mount`, que recebe o tipo do sistema de arquivos (`userfs`), o ponto de montagem e os descritores de arquivos do *pipe*. Assim, `muserfs` indica ao núcleo do UserFS que um novo sistema de arquivos será montado e passará a responder à requisições pelo *pipe* recentemente aberto. Este processo é mostrado na Figura 4.3b.

O VFS do LINUX então cria uma estrutura `superblock`, que representa um sistema de arquivos montado no VFS do LINUX (seção 4.1.3) e a associa ao `inode` que representa seu ponto de montagem. A operação `mount` do UserFS é então chamada pelo VFS. A operação `mount` do UserFS, requisita ao processo do usuário, via *pipe* e utilizando a requisição `up_mount` do protocolo, um identificador (*handle*) que representa o inode raiz do sistema de arquivos. Embora para o núcleo do UserFS a requisição `up_mount` só necessite retornar o identificador do inode raiz, a operação correspondente no processo geralmente realiza outras tarefas de inicialização, particulares à cada implementação.

A segunda requisição feita pelo UserFS é um `up_iread` com o identificador do inode raiz como parâmetro. Esta requisição retorna os atributos do inode. Depois disso, o núcleo manda uma série de requisições do tipo `up_enquire`, uma para cada operação do protocolo. O processo de usuário responde às operações que este suporta com 0 e às que não suporta com `ENOSYS` (constante padrão do UNIX). Após todas as operações serem testadas, o sistema de arquivos estará montado e pronto para responder a outras requisições suportadas.

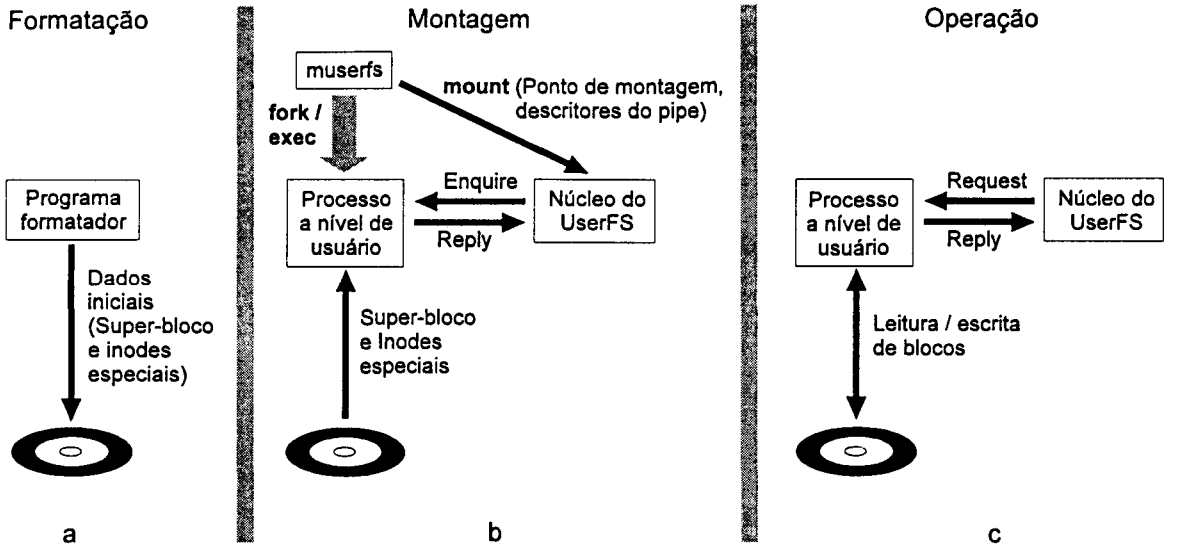


Figura 4.3: Utilização de um sistema de arquivos do tipo `userfs`.

Quando alguma aplicação realiza uma operação em um sistema de arquivos do tipo `userfs`, o VFS a dirige para o núcleo do UserFS, que na verdade corresponde ao nível de implementação do tipo de sistema de arquivos `userfs`. O núcleo então repassa esta requisição ao processo de usuário. Este realiza a operação e devolve o resultado ao UserFS, que retorna o controle à camada VFS do LINUX.

4.2.4 Protocolo

A comunicação entre o processo de usuário e o UserFS é muito simples. Basicamente, do lado do processo, o código deve desempacotar a mensagem, identificar qual requisição está sendo realizada, executar as operações do sistema de arquivos que correspondem à esta requisição, empacotar o resultado e responder ao núcleo. Assim, o esqueleto básico do processo fica muito semelhante em todos os sistemas de arquivos desenvolvidos com a ferramenta. Geralmente, os processos possuem um laço contínuo que só termina quando uma operação `up_umount` é recebida, ou quando um EOF é lido do *pipe*, o que significa que o núcleo não enviará mais nenhuma requisição.

Toda a comunicação entre o núcleo do UserFS e os processos de usuário é feita através de um protocolo independente de máquina, que utiliza ordenação de bytes de rede e tipos de dados próprios, análogos aos tipos de C. Estes tipos, assim como tipos mais complexos como agregados, são definidos utilizando-se uma linguagem de definição de dados própria da aplicação.

Todo pacote do protocolo possui um cabeçalho, que guarda a versão do protocolo, um número de seqüência, o tipo da operação, o tipo do pacote e o tamanho da área de dados do pacote. O tipo do pacote pode ser `up_enquire`, `up_request` ou `up_reply`. *Enquires* e *requests* são sempre enviados do núcleo para o processo e apenas *replies* são enviados no sentido contrário. Os cabeçalhos dos pacotes *reply* possuem também um campo de erro. Se a operação foi realizada com sucesso, este campo possui 0, senão, possui um código de erro padrão do UNIX. O campo de erro também é utilizado nas respostas dos *enquires*, onde o processo de usuário retorna 0 ou `ENOSYS` ao núcleo. Quando um processo responde a uma requisição, o número de seqüência do *reply* deve ser o mesmo do *request* ao qual este está associado.

Os tipos de pacote que podem possuir alguma informação na área de dados são *requests* e *replies*. Os códigos em C e C++ para "serializar" e "desserializar" as estruturas que contém os parâmetros e os resultados das operações são gerados automaticamente por uma aplicação. Esta aplicação interpreta arquivos de definições feitos utilizando-se a linguagem de definição de tipos específica do protocolo. Assim, novas operações podem ser facilmente adicionadas ao protocolo e este pode ser utilizado em outros sistemas e linguagens, alterando-se apenas a aplicação que gera o código de "serialização" e "desserialização".

As operações que o protocolo implementa cobrem as principais operações existentes no VFS do LINUX (Figura 4.1) e estão sumariadas na Tabela 4.1. Estas são utilizadas no campo tipo da operação do cabeçalho, tanto em *enquires* quanto em *requests*. As operações do VFS que não estão no protocolo são tratadas pelo próprio núcleo do UserFS.

Entre os processos de usuário e o núcleo do UserFS, os inodes são referenciados por identificadores (*handles*), que são números de 32 bits sem sinal. O uso de identificadores permite uma referência uniforme e única para qualquer tipo de sistema de arquivos. Assim, cabe a cada processo gerar um identificador único para cada inode exigido pelo núcleo e, quando mais tarde receber este identificador novamente, realizar o mapeamento para o inode correto. As operações que recebem um nome de arquivo e devolvem um identificador são `up_lookup`, `up_open` e `up_create`. As demais operações sobre inodes recebem identificadores como parâmetros de entrada.

As operações do protocolo `up_create` e `up_unlink` substituem mais de uma operação do VFS do LINUX. Por exemplo, `up_create` pode ser utilizada para arquivos comuns, correspondendo a `create` do VFS, para arquivos diretório, correspondendo a `mkdir`, ou para arquivos especiais, correspondendo a `mknod`. Esta estratégia é utilizada para diminuir o número de mensagens entre o núcleo e o processo de usuário e, em conseqüência, diminuir o número de trocas de contexto do sistema. Isso ocorre pois as interfaces para `mkdir` e `mknod`, tanto no LINUX quanto no VFS original, não retornam o inode do arquivo recentemente criado. Assim, para encontrá-lo seria necessária uma operação `lookup` com o nome dado ao arquivo criado. O UserFS evita isso retornando o identificador do inode para qualquer um dos tipos de arquivo na operação `up_create`. O processo de usuário fica encarregado de saber a que tipo de arquivo estas operações estão se referindo. Em `up_create`, por exemplo, isso é feito averiguando o modo de criação do arquivo, que é passado como parâmetro da requisição.

Operação do UserFS		Descrição da operação
0	`up_create`	Cria um arquivo (normal, diretório, *pipe*, etc.).
1	`up_lookup`	Procura por um nome de arquivo em um diretório.
2	`up_open`	Abre um arquivo.
3	`up_close`	Fecha um arquivo.
4	`up_read`	Lê de um arquivo.
5	`up_write`	Escreve em um arquivo.
6	`up_truncate`	Modifica o tamanho de um arquivo.
7	`up_fsync`	Grava os blocos modificados de um arquivo.
8	`up_readdir`	Lê uma entrada de diretório.
9	`up_link`	Cria uma referência de arquivo em um diretório.
10	`up_unlink`	Remove uma referência de arquivo de um diretório (arquivo normal, diretório, *pipe*, etc.).
11	`up_symlink`	Cria um *link* simbólico.
12	`up_readlink`	Lê um *link* simbólico.
13	`up_followlink`	Resolve um caminho através de um *link*.
14	`up_mount`	Devolve um identificador para o inode raiz e realiza operações específicas para cada sistema de arquivos.
15	`up_umount`	Salva as estruturas modificadas do sistema de arquivos.
16	`up_iread`	Lê os conteúdos de um inode.
17	`up_iwrite`	Grava os conteúdos de um inode.
18	`up_iput`	Libera um inode.
19	`up_statfs`	Retorna atributos do sistema de arquivos.
20	`up_permission`	Checa permissões sobre um arquivo.
21	`up_rename`	Renomeia um arquivo.
22	`up_multreaddir`	Lê diversas entradas de um diretório de uma só vez.
23	`up_notify_change`	Notifica mudanças nos atributos do inode.
24	`up_inode_valid`	Verifica se um inode é válido.

Tabela 4.1: Operações do protocolo do UserFS.

4.2.5 Instalação da ferramenta

A instalação do pacote é simples, mas exige uma compilação inicial do núcleo e da aplicação `muserfs`. Todo o código da ferramenta está disponível para ser analisado e algumas

implementações de sistemas de arquivos experimentais são incluídos na distribuição. É instalada ainda uma biblioteca orientada a objetos, chamada `libuserfs`, escrita em C++, que implementa classes abstratas primitivas para sistemas de arquivos, inodes e arquivos abertos. Os métodos destas classes devem ser implementados segundo as necessidades de cada sistema de arquivos experimental.

Capítulo 5

Uma proposta de um LFS para o LINUX

Na ocasião do início deste projeto, em maio de 1996, não encontramos referências sobre nenhum trabalho que se dedicasse a realizar uma implementação do LFS no LINUX. Assim, portar este sistema de arquivos seria uma grande chance de aprender mais sobre o próprio LFS e o LINUX e contribuir para o desenvolvimento de ambos. Após o início do projeto, encontramos o anúncio de um trabalho que se dedicava a portar o LFS para o LINUX [COO98]. Na época, o trabalho ainda estava em seu início e pouca coisa havia sido feita. Até hoje este projeto está em andamento e, embora alguns progressos tenham sido feitos, não parece estar próximo do seu final.

O trabalho de portar um sistema de arquivos integralmente para um novo sistema operacional é grande e depende de diversas habilidades, em especial um profundo conhecimento do núcleo do sistema operacional em questão, de seu subsistema de arquivos e do modelo de sistema de arquivos que se quer portar. Ao iniciarmos o projeto, não tínhamos conhecimento suficiente em nenhum destes três aspectos mencionados para rescrever o LFS diretamente no núcleo do LINUX. Desenvolver estas habilidades também tomaria um tempo razoável, como de fato tomou, e atrasaria o término do projeto, visto que a implementação seria realizada por apenas uma pessoa.

Tínhamos na época do início do projeto o conhecimento da existência da ferramenta UserFS (seção 4.2), que possibilita a construção de sistemas de arquivos experimentais a nível de usuário. Uma primeira análise do UserFS mostrou que este seria simples de utilizar e facilitaria a implementação do LFS. Assim, foi tomada a decisão de utilizar prototipação para desenvolver o sistema de arquivos, com a ferramenta UserFS servindo de ligação entre o núcleo do LINUX e o programa que conteria o código do LFS. A escolha de utilizar o UserFS nos permitiu também analisar o impacto do desenvolvimento de sistemas de arquivos com o auxílio de uma ferramenta de prototipação. Resolvemos então chamar o novo sistema de arquivos de **ULFS**, ou seja "User LFS", e o programa que desenvolvemos para manipulá-lo de **`ulfs`**. Além do `ulfs`, implementamos uma série de aplicativos, dentre os quais um formatador e vários outros, que compreendem testes de correção e desempenho do sistema.

A primeira parte deste capítulo descreve a organização do sistema ULFS e as estruturas de dados que o compõem. Depois, são mostradas as experiências iniciais que

tivemos com o UserFS, que compreendem o desenvolvimento de um pequeno sistema de arquivos, do qual retiramos várias idéias para o programa `ulfs`. Mais adiante, mostraremos os passos seguidos para desenvolver o programa `ulfs`, seu funcionamento e os testes para compará-lo ao Ext2 e avaliar o seu desempenho. O capítulo encerra-se com as conclusões do projeto e com algumas possíveis extensões que podem ser realizadas em trabalhos futuros.

5.1 Organização do sistema ULFS

Nesta seção são descritas as principais estruturas encontradas em um sistema de arquivos ULFS, a disposição física destas estruturas no disco e sua organização na memória principal. A separação entre a estrutura do ULFS e a implementação do protótipo `ulfs` feita nesta dissertação é apenas didática. Na verdade, durante o projeto, diversas mudanças foram realizadas tanto nas estruturas quanto nos algoritmos do programa, em um processo iterativo.

Diversas decisões de projeto foram tomadas durante o desenvolvimento do ULFS. Algumas destas dizem respeito à implementação do programa `ulfs`. Outras foram motivadas pelas diferenças entre o Sprite LFS e o BSD LFS. O nosso sistema assemelha-se mais ao BSD LFS, pois este corrige algumas deficiências da implementação anterior. Várias características do nosso sistema, como o uso do `ifile` para armazenar a tabela de utilização de segmentos e o mapa de inodes, a replicação do superbloco, a organização dos segmentos e o *checksum* do sumário de segmento foram retiradas deste modelo mais recente.

5.1.1 Superbloco e checkpoint

No ULFS existe uma diferença entre o superbloco em disco e em memória. Em disco, a estrutura que representa o superbloco é `ulfs_static_info` (seção A.1.1) que guarda as informações que não mudam com o uso do sistema de arquivos. Após cada cópia deste "super bloco estático", existe uma região de *checkpoint*, representada por uma estrutura do tipo `ulfs_checkpoint` (seção A.1.2). Esta guarda as informações que determinam o estado atual do sistema de arquivos, como a posição do inode do `ifile`, o número do segmento corrente, o número do bloco dentro do segmento corrente que representa o início do segmento parcial atual e uma marca de tempo. Existem duas cópias do superbloco em disco: uma no início e outra no fim da partição. A Figura 5.1 mostra a organização geral de um sistema ULFS. Alguns campos do superbloco foram omitidos, pois podem ser conseguidos através dos outros campos existentes.

O número máximo suportado de blocos de dados no sistema de arquivos é próximo de 4G. O maior endereço de um bloco de dados de um arquivo é 2^{31}, ou seja 2G endereços, mas na verdade o tamanho máximo de um arquivo é limitado pelo tamanho do bloco de dados. O primeiro bloco logo após o superbloco possui o endereço 0 e seus consecutivos são

endereçados em ordem crescente até o último bloco do último segmento. As duas cópias dos superblocos e das regiões de *checkpoint* não são consideradas como espaço de endereçamento de blocos.

Decidimos também utilizar uma combinação de blocos de dados grandes (geralmente 4KB) e fragmentos, ao contrário do que faz o Ext2, no qual o tamanho padrão do bloco é 1KB e não existem fragmentos. O uso de fragmentos permite diminuir o número médio de blocos alocados por arquivo, diminuindo o espaço utilizado pelos metadados e aumentando a velocidade de operações de E/S sobre arquivos grandes, sem aumentar a fragmentação interna do espaço em disco. Infelizmente, a implementação de fragmentos ainda não foi realizada no protótipo. Embora as estruturas internas do `ulfs` já suportem o uso de fragmentos, os algoritmos do sistema ainda não conseguem manipulá-los.

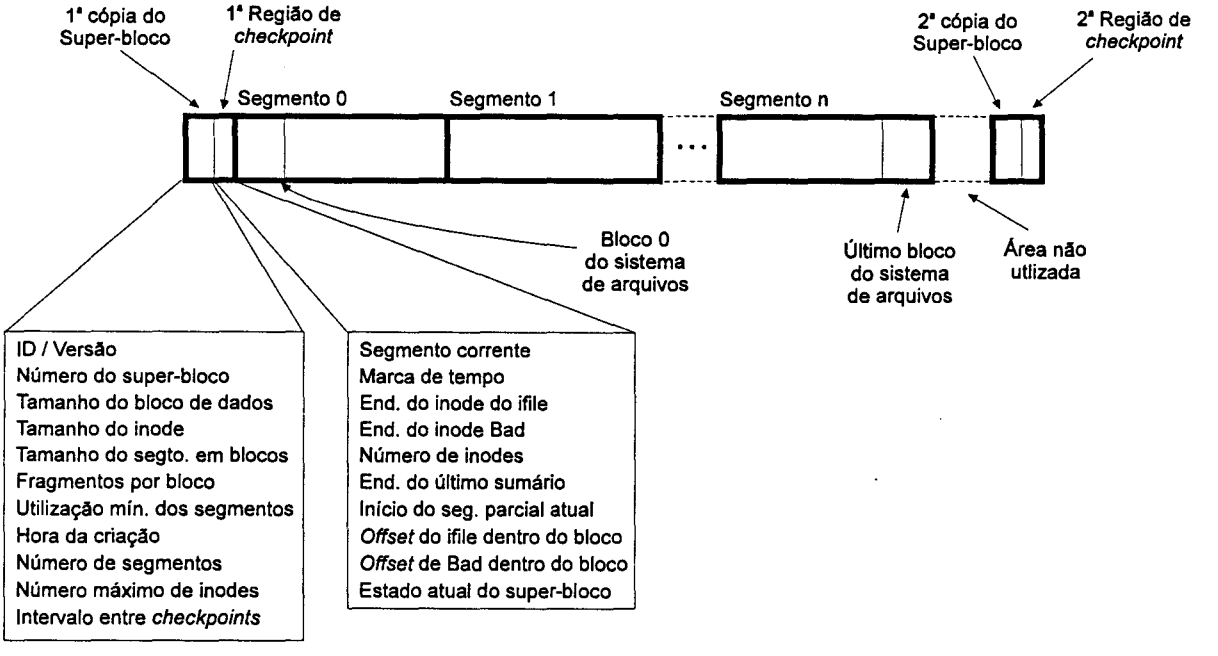


Figura 5.1: Organização de uma partição ULFS.

O superbloco é representado em memória por uma estrutura do tipo `ulfs_super_block` (seção A.1.3). Esta possui um campo chamado `static_sb`, do tipo `ulfs_static_info`, e outro, chamado `check_reg`, do tipo `ulfs_checkpoint`. Estas estruturas são preenchidas com dados vindos do disco quando o sistema de arquivos é montado. Existe ainda um campo, `cur_seg`, do tipo `ulfs_segment` (seção A.4.1), que representa o segmento corrente em memória. O segmento corrente é responsável por armazenar os blocos sujos que devem ser gravados no *log*, mas que devido ao modelo do LFS, ainda não foram ordenados e não possuem endereço definitivo (seção 5.1.5). O superbloco guarda ainda os inícios das tabelas de *hashing* de inodes e de blocos, apontadores para os inodes especiais do sistema de arquivos (`bad`, `ifile` e `root`) e um apontador, `files`, para uma lista de estruturas `ulfs_file`, que guarda os arquivos abertos, além de outras informações de controle das estruturas em memória.

As tabelas de *hashing* do superbloco consistem em dois vetores de apontadores para estruturas do tipo `ulfs_inode` e `ulfs_block`, respectivamente, nos quais cada posição marca o início de uma lista duplamente ligada de estruturas. Atualmente, estes vetores possuem 512 posições cada. A função de *hashing* utilizada para ambas as tabelas é praticamente a mesma e consiste na operação módulo do número do elemento que está sendo manipulado (inode ou bloco) pelo tamanho da tabela. No caso dos inodes, a última posição da tabela de *hashing* é reservada para a lista de estruturas `ulfs_inode` que estão livres, ou seja, que podem ser reutilizadas para outros inodes. Os blocos da *cache* que são liberados são apagados da memória, pois ocupam um espaço muito maior em relação aos inodes.

O sistema de arquivos e sua *cache* são auto contidos no superbloco, ou seja, qualquer outra estrutura em memória ou em disco pode ser encontrada a partir deste.

5.1.2 Cache de blocos e metadados

A *cache* de dados é essencial ao LFS, pois este modelo de sistema de arquivos está centrado justamente no seu uso para coletar vários blocos, de dados e metadados, que serão descarregados ao mesmo tempo em segmentos no disco. Ainda, os blocos de dados e metadados destes segmentos devem ser ordenados antes de serem gravados efetivamente no disco, o que exige uma flexibilidade muito grande da *cache*.

Por estarmos trabalhando em nível de usuário, não tínhamos acesso à *cache* do VFS do LINUX. Não era possível nem mesmo copiar os algoritmos que implementam a *buffer cache*, pois estes são totalmente interligados e dependentes do subsistema de memória. Implementar uma *cache* de dados implica em controlar a alocação e liberação de páginas de memória, implementar políticas de descarga de blocos sujos – normalmente baseadas em algoritmos do tipo *Last Recently Used* – controlar a concorrência pelos objetos em *cache* (como *buffers*, inodes e entradas de diretório) e manter a coerência entre os dados salvos em disco e os que ainda estão em memória, entre outras tarefas. Realizar este trabalho não é simples e demanda muitos testes para assegurar a correção dos algoritmos.

Assim, para o `ulfs`, decidimos implementar uma *cache* simples, com uma organização bem mais rígida e capacidades muito mais modestas do que as esperadas por um sistema de arquivos real, mas que atende perfeitamente às nossas necessidades. Os elementos que são colocados na *cache* são o superbloco, que inclui uma região de *checkpoint*, os inodes e os blocos, tanto de dados quanto de metadados.

Uma das metas ao desenvolvermos o ULFS era manter a organização da sua *cache* parecida com a do LINUX, para que o trabalho de portar os algoritmos para a camada VFS fosse facilitado. Como já foi visto anteriormente (seção 4.1.2), a *buffer cache* do LINUX indexa os *buffers* pelo seu endereço no dispositivo e não pela sua posição lógica dentro do arquivo para o qual está alocado. Este esquema de indexação causa um problema que afeta em especial as implementações do LFS.

A organização da *buffer cache* do LINUX baseia-se no fato de que cada bloco de dados de um arquivo possui um endereço fixo no disco, invariável desde o momento em que é alocado para o arquivo até a hora em que é liberado, quando o arquivo é apagado ou truncado. Quando um bloco é lido, por exemplo, é associado a este um *buffer*, que é inserido na tabela de *hashing* de *buffers* segundo o endereço físico do bloco e do dispositivo (partição) a que pertence. Se o bloco é alterado, o *buffer* que o contém é marcado como sujo para que seja novamente gravado no disco.

No LFS, entretanto, os blocos de um arquivo mudam de posição dentro do *log* quando são alterados ou quando os segmentos aos quais pertencem são limpos (seção 3.1.4). Assim, quando um bloco é alterado, um novo endereço deve ser associado a este e o *buffer* que o representa deve mudar de posição na tabela de *hashing*, para que as rotinas internas do sistema de arquivos possam encontrá-lo. Mas o LFS não associa endereços aos blocos sujos até que estes estejam prontos para serem gravados no disco. Isso é feito pois os segmentos são ordenados imediatamente antes de serem salvos e não se sabe de antemão que posição um bloco ocupará dentro do segmento. Além disso, mais de um segmento pode ser necessário para gravar todos os *buffers* sujos que estão em memória. Desta forma, uma vez que o bloco é sujo e seu *buffer* é retirado da tabela de *hashing*, não há como inseri-lo novamente na tabela, pois este não possui mais um índice válido.

Uma saída para o problema acima seria manter o endereço do bloco inalterado até que o segmento ao qual pertence fosse ordenado, quando receberia efetivamente um novo endereço. Porém, imagine o caso em que o segmento ao qual o bloco em questão pertencia anteriormente tenha sido escolhido para ser limpo antes do novo segmento ser escrito. Mesmo que o código do sistema de arquivos impeça que o bloco alterado seja substituído por sua versão mais antiga, uma vez que ambos possuem o mesmo endereço, a utilização do espaço em disco anteriormente ocupado pelo bloco não seria permitida até que o bloco fosse salvo. Isso acontece pois seu *buffer* continua com o mesmo endereço antigo.

Em nosso protótipo, mantivemos a indexação dos blocos pela sua posição no dispositivo. A adaptação que realizamos é simples e consiste em reservar os endereços mais altos do espaço de endereçamento de blocos para os blocos que não têm endereços determinados, ou seja, os bloco sujos da *cache*. Desta forma, não há conflitos entres estes endereços e os endereços dos blocos na partição. Assim, se o endereço dos blocos for um número de 32 bits sem sinal (que é o valor utilizado atualmente no `ulfs`), teríamos 4G

endereços de blocos. Porém, se os últimos 2^{16} endereços forem reservados para endereços temporários, não será mais possível endereçar os 4G, embora o número de endereços permaneça bem próximo de 2^{32}. Se considerarmos um bloco de 4KB teríamos uma partição de mais de 8TB como limite máximo para o sistema de arquivos. Esta solução também limita os segmentos a possuírem no máximo 65536 blocos, que na verdade é um número razoavelmente alto, se considerarmos também que o tamanho de um segmento é normalmente abaixo de um megabyte e que o tamanho do bloco de dados mais indicado é 4KB ou acima disso.

5.1.3 Blocos

Blocos são representados na memória por estruturas do tipo `ulfs_block` (seção A.3.2), que correspondem aos *buffers* das *caches* tradicionais de blocos. Esta estrutura possui um campo `type`, do tipo `ulfs_blk_type` (seção A.3.1), importante para o LFS pois cada tipo de bloco possui um tratamento especial na ordenação do segmento. Outros campos importantes são o endereço lógico do bloco no sistema de arquivos e um apontador para a área de dados do bloco.

Quando um novo bloco é lido, este vai para a tabela de *hashing* de blocos. Caso seja modificado, o bloco irá para a estrutura `cur_seg` do superbloco, que representa o segmento corrente, onde permanecerá até que seja-lhe associado um endereço de disco. O endereço definitivo é associado durante a ordenação do segmento, quando cada inode em *cache* e, em conseqüência, seus blocos são analisados. Assim, quando a posição definitiva do bloco no segmento é determinada, este é marcado como limpo, retirado da estrutura `cur_seg` e recolocado na tabela de *hashing* de blocos. A localização do bloco no *hashing* depende de seu novo endereço.

Caso o espaço em memória para novos blocos se esgote, um bloco da tabela de *hashing* é escolhido para ser aproveitado novamente. A escolha se dá aleatoriamente, pois a *cache* não implementa controle sobre o tempo de último acesso ao bloco, o que normalmente determina qual bloco será reutilizado. Como todos os blocos da tabela de *hashing* estão limpos, qualquer um pode ser escolhido.

5.1.4 Inodes e files

No `ulfs`, os inodes são representados em memória por estruturas `ulfs_inode` (seção A.2.2). Esta estrutura possui um apontador para uma estrutura do tipo `ulfs_disk_inode` (seção A.2.1), que é preenchida com os dados vindos do disco.

O inode do ULFS possui 12 apontadores diretos, um apontador indireto e um apontador duplamente indireto. Existe um mapa de bits em cada inode para indicar seus fragmentos, em número máximo de 8, embora o `ulfs` não consiga ainda manipular

fragmentos. Atualmente, o tamanho do inode em disco do ULFS é de 100 bytes. Como no Sprite LFS, o inode em disco não contém a hora do último acesso, que fica armazenado no mapa de inodes. Já a estrutura `ulfs_inode` possui um campo chamado `atime`, que representa este atributo e é retirado do mapa de inodes quando o inode é lido do disco para a *cache*.

O sumário de segmento deve associar cada bloco do segmento ao seu inode correspondente. Assim, tanto os blocos de dados quanto os blocos de apontadores devem possuir índices para que sejam identificados unicamente. Os blocos de dados são numerados logicamente a partir de 0, até um valor máximo de 2^{31}-1. Os blocos de apontadores são numerados da seguinte forma:

- O bloco de apontadores para blocos indiretos recebe o valor -1.
- O bloco de apontadores duplamente indiretos recebe o valor -2.
- O primeiro bloco de apontadores que é apontado pelo bloco duplamente indireto recebe o valor -3.
- O n-ésimo bloco de apontadores que é apontado pelo bloco duplamente indireto recebe o valor -(n+3).

Arquivos abertos são representados por estruturas do tipo `ulfs_file` (A.2.3). Os principais campos desta estrutura são a posição do apontador de leitura/gravação, `f_pos`, e o inode ao qual o arquivo se refere, `f_inode`. As estruturas `ulfs_file` são guardadas em uma lista que se inicia no superbloco do sistema de arquivos e não possuem estruturas correspondentes em disco. Estas estruturas foram criadas para manter a compatibilidade com o VFS do LINUX. Algumas das operações definidas na sua interface, tais como `read`, `write` e `close`, são realizadas sobre estruturas `file` e não sobre `inode`.

Ao contrário do que ocorre no VFS, inclusive do LINUX, as estruturas `ulfs_inode` e `ulfs_file` não possuem apontadores para as suas operações. Estes apontadores não são necessários pois o protótipo manipula apenas inodes e arquivos do sistema ULFS.

5.1.5 Segmento

Em nosso protótipo, existe uma estrutura no superbloco, chamada `cur_seg`, do tipo `ulfs_segment` (seção A.4.1), que consiste no segmento atualmente sendo alterado, chamado de segmento corrente. Esta possui um vetor de apontadores para estruturas `ulfs_seg_entry`, que guardam blocos de dados dos arquivos (incluindo dos diretórios e do `ifile`) e metadados, como o sumário do segmento e blocos de inodes. Este vetor é alocado na iniciação do sistema e possui o tamanho de um segmento. São armazenadas também nesta estrutura informações de controle, como o número do segmento atualmente sendo escrito, o número de blocos sujos no segmento e o deslocamento do primeiro bloco do segmento parcial dentro do segmento corrente.

Uma estrutura `ulfs_seg_entry` (seção A.4.2) possui um apontador, `blk`, para uma estrutura `ulfs_block`, um apontador, `owners`, para estruturas do tipo `own` – que identifica o(s) inode(s) dono(s) do bloco – um *flag* chamado `fragmented`, que indica se o bloco é fragmentado, e um campo `next`, que indica o próximo bloco em uma das listas do segmento (blocos livres, blocos de sumário e blocos de inodes).

Quando o vetor de estruturas `ulfs_seg_entry` é alocado, `cur_seg.blks` passa a apontar para este e o ULFS verifica se o sistema sendo montado possui fragmentos. Em caso afirmativo, são alocadas tantas estruturas `own` por posição do vetor quanto o número de fragmentos suportados por bloco. Caso contrário, apenas uma estrutura `own` é alocada por posição do vetor.

É importante para o sistema saber se um determinado bloco é fragmentado e quais são os inodes que possuem fragmentos neste bloco, para indicar corretamente esta situação no sumário de segmento. A estrutura `own` indica o inode, sua versão e a posição lógica do bloco dentro do arquivo. A posição da estrutura no vetor `owners` corresponde à posição do fragmento dentro do bloco. Assim, se um inode ocupa os fragmentos 3 e 4 de um bloco, `owners[3]` e `owners[4]` vão se referir a este mesmo inode.

O *flag* `fragmented` só possui valor verdadeiro quando o bloco estiver sendo ocupado por mais de um inode.

O campo `next` é utilizado para encadear três listas diferentes do segmento corrente. A primeira é a lista de blocos livres, cuja cabeça está em `cur_seg.freeblk_lst`. Inicialmente, quando o sistema é montado, todos os blocos do segmento estão livres. As duas outras listas são as de blocos de sumário, `curseg.summblk_lst`, e de blocos de inodes, `cur_seg.inoblk_lst`. Os blocos de sumário e de inodes vão sendo alocados no segmento corrente conforme sejam necessários, ou seja, conforme o espaço do bloco previamente alocado tenha se esgotado. Um bloco estará sempre em apenas uma destas listas, assim, apenas um campo `next` é necessário. Os blocos de dados não estão em nenhuma fila, mas mesmo assim podem ser acessados pela posição que ocupam no vetor `cur_seg.blks`.

Apenas blocos modificados e blocos recentemente alocados são colocados no segmento corrente. Estes blocos recebem endereços especiais, fora do domínio de endereçamento normal do sistema de arquivos (seção 5.1.2). Quando é necessário inserir um bloco no segmento corrente, o sistema procura uma posição livre, `p`, no vetor, faz o apontador `cur_seg.blks[p].blk` apontar para o bloco, cria um endereço especial somando o menor valor de endereços especiais (definido em uma constante do `ulfs`) à posição `p` encontrada e faz o endereço do bloco igual a este endereço especial. Da mesma forma, quando o sistema deseja encontrar um destes blocos sujos, este traduz o seu endereço para uma posição dentro do vetor de blocos.

Ao final da ordenação do segmento, pode ocorrer que um ou mais blocos de sumário ou de inodes pré-alocados não tenha sido utilizado. Isso pode ocorrer, por exemplo, caso exista algum inode *busy* no segmento que não pode ser gravado (veja a seção 5.3.2). Se isso ocorrer, estes blocos são inseridos novamente na lista de blocos livres. Felizmente, é pouco comum sobrarem blocos no final da ordenação e a maioria dos segmentos são salvos com utilização total.

Em disco, um segmento possui a organização mostrada na Figura 5.2. Em primeiro lugar é alocado um bloco para o sumário do segmento, seguido de um bloco de inodes. A ordem em que os blocos de cada inode são postos no segmento é a seguinte:

1. `1` bloco de apontadores para blocos indiretos de dados
2. `1` bloco de apontadores para blocos de apontadores duplamente indiretos
3. `n` blocos de apontadores para blocos duplamente indiretos de dados
4. `n` blocos diretos de dados
5. `n` blocos indiretos de dados
6. `n` blocos duplamente indiretos de dados

É claro que estes elementos aparecem apenas quando forem necessários. Por exemplo, haverá um bloco de apontadores para blocos indiretos apenas se algum bloco de dados indireto do inode for gravado neste segmento. Os blocos dos inodes são organizados desta forma pois os apontadores mais altos estão sempre posicionados antes dos apontadores de nível mais baixo e dos blocos de dados. Como o acesso aos blocos de nível mais baixo está sempre condicionado ao acesso do apontador de nível mais alto, este pode beneficiar-se de algum algoritmo de leitura antecipada que venha a ser implementado.

Mais de um bloco de sumário pode ser encontrado em um mesmo segmento. Cada bloco de sumário no *log* corresponde a um segmento parcial. Um novo bloco de sumário é alocado quando o sumário atual está cheio – ou seja, não há mais espaço para novas entradas – e ainda existe espaço físico no segmento corrente, quando o intervalo de tempo entre *checkpoints* foi atingido ou quando uma chamada `fsync` for realizada. As duas últimas situações forçam o sistema a descarregar o segmento que está em memória.

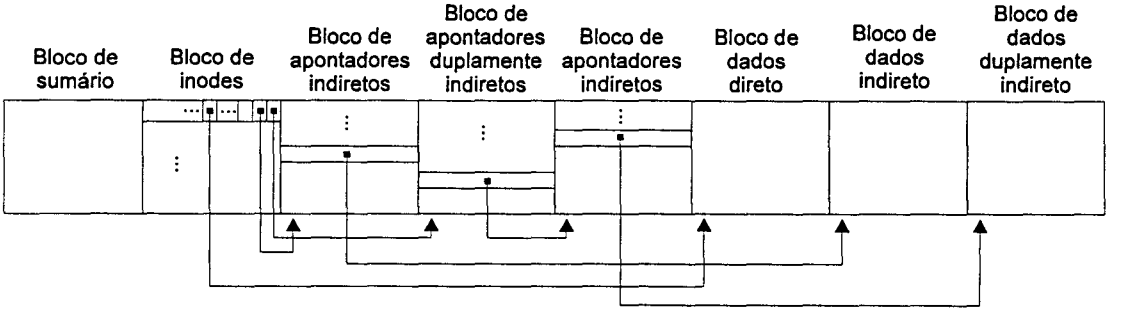


Figura 5.2: Organização dos blocos no segmento do ULFS.

5.1.6 Sumário de segmento

Nem o Sprite LFS nem o BSD LFS implementam fragmentos. Assim, o sumário de segmento destes dois sistemas não possui nenhuma informação acerca de fragmentos. No nosso caso, é essencial que o sumário identifique quais os blocos fragmentados e a quais inodes estes fragmentos pertencem, pois o mecanismo de limpeza depende destas informações para copiar os dados válidos dos segmentos.

O sumário do ULFS é muito parecido com o do Sprite LFS (seção 3.1.3) mas possui um cabeçalho inspirado no modelo do BSD LFS (seção 3.2.2). A estrutura que representa o cabeçalho do sumário é `ulfs_summ_header` (seção A.5.1). Neste cabeçalho encontra-se um *checksum* computado sobre os dados dos blocos do segmento e um outro, feito sobre o próprio sumário. Estes dois *checksums* são utilizados para verificar a consistência dos dados do segmento, no caso de falhas do sistema. O cabeçalho ainda guarda o número de blocos sumariados, a hora de criação do sumário e um par de apontadores para os sumários imediatamente anterior e posterior no *log*.

Após o cabeçalho, o sumário possui uma seqüência de entradas `ulfs_summ_entry` (seção A.5.2), que possuem o número do inode ao qual o bloco pertence, a posição lógica do bloco no inode e o número de versão do inode.

No algoritmo de ordenação de segmentos, quando um bloco fragmentado é encontrado, uma entrada com um código especial no campo `inode` é inserida no sumário. Isso indica que as próximas `n` entradas do sumário, onde `n` é o número de fragmentos de um bloco (conforme indicado no superbloco), vão se referir a um mesmo bloco de dados. Assim, para cada inode que possui um ou mais fragmentos neste bloco, uma ou mais entradas serão adicionadas neste bloco. Os fragmentos que não possuem informações válidas, ou seja, não estão alocados para nenhum inode, são indicados por entradas com o valor `ULFS_INVALID_INODE` no campo inode.

5.1.7 Ifile

O `ifile` do `ulfs` possui uma estrutura muito próxima do `ifile` do BSD LFS (seção 3.2.3). No primeiro bloco do arquivo existe um cabeçalho, formado por uma estrutura `ulfs_ifile_head` (seção A.6.1), que possui informações que o *cleaner* utiliza e informações de controle sobre o próprio `ifile`. Estas informações incluem o bloco lógico do arquivo onde o mapa de inodes se inicia e o número do primeiro inode livre da lista de livres. Ainda no primeiro bloco do arquivo, inicia-se a tabela de utilização de segmentos, formada por estruturas do tipo `ulfs_sut_entry` (seção A.6.2). A tabela de utilização de segmentos possui tamanho fixo e pode necessitar de um ou mais blocos de dados do `ifile`, dependendo do tamanho do bloco do sistema de arquivos e do número total de segmentos.

O mapa de inodes inicia-se no primeiro bloco logo após a tabela de utilização de segmentos e é composto por estruturas do tipo `ulfs_imap_entry` (seção A.6.3).

Inicialmente, quando o `ifile` é criado, é alocado um bloco de dados para o mapa de inodes. Este é preenchido com estruturas `ulfs_imap_entry` que são colocadas na lista de inodes livres. Quando todas as entradas deste bloco forem utilizadas, a lista de inodes livres estiver vazia e novos inodes forem necessários, um novo bloco é alocado e suas entradas são colocadas na lista de inodes livres também. Esta operação se repete enquanto o `ifile` puder crescer. Como o número máximo de blocos em um arquivo é 2^{31} e cada entrada do mapa de inodes ocupa apenas 16 bytes (o que dá 256 entradas em um bloco de 4KB), o número de inodes suportado pelo `ifile` é praticamente inesgotável. A Figura 5.3 traz a organização do `ifile` do `ulfs`. Nesta, a tabela de utilização de segmentos ocupa `n` blocos de dados e o mapa de inodes encontra-se com `m` blocos de dados. Note que o último bloco do mapa de inodes está sendo completamente utilizado, pois as entradas não assinaladas para inodes são postas na lista de livres.

No ULFS, diferentemente do BSD LFS, o tempo do último acesso ao arquivo é mantido no mapa de inodes, e não no inode. Assim, o maior problema atribuído a esta implementação, que era a excessiva movimentação dos inodes dentro do *log*, fica resolvido. Na tabela de utilização de segmentos, os *flags* indicam o estado do segmento, que pode ser "limpo", "usado", "segmento corrente" ou "sendo limpo". O *flag* "sendo limpo" é utilizado no processo de limpeza e evita que este segmento seja escolhido como o próximo segmento atual ou seja escolhido para ser limpo novamente, caso existam outros *cleaners* ativados ao mesmo tempo.

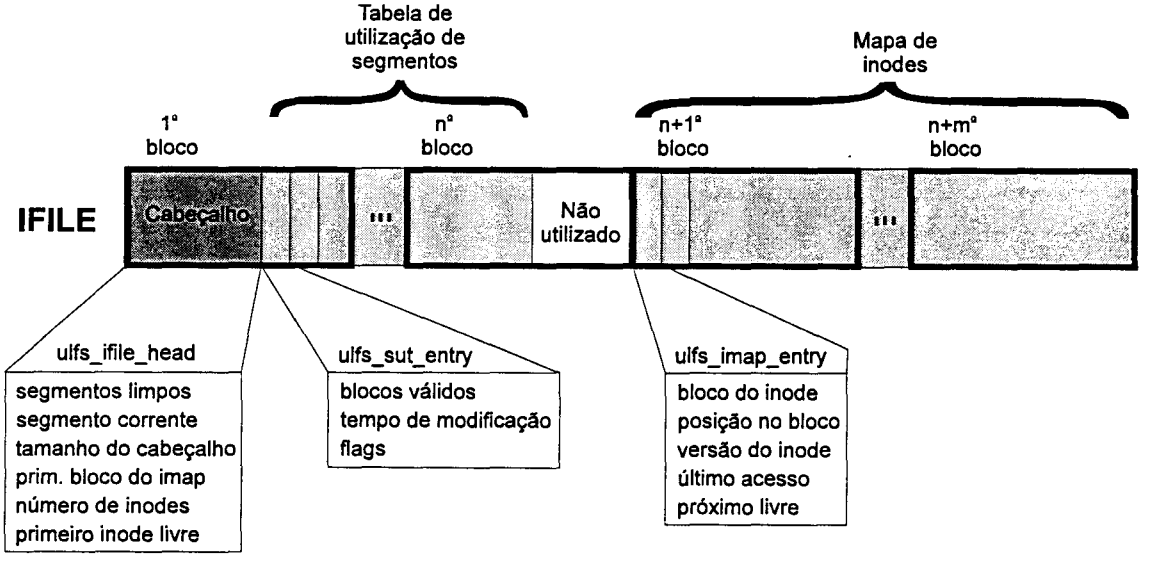


Figura 5.3: O ifile do ulfs.

O mapa de inodes, e em conseqüência o ifile, guarda a posição de todos os inodes do disco, como mostrado na Figura 5.4. Isso inclui os inodes especiais do sistema de arquivos, que são a lista de blocos ruins, `bad`, o diretório raiz, `root`, e próprio `ifile`. Na verdade, quando necessita encontrar `ifile` ou `bad`, o `ulfs` recorre à região de *checkpoint* que está em memória, para evitar procurar pela entrada correspondente do mapa de inodes. Todos os demais inodes são encontrados pelo mapa de inodes.

5.2 Desenvolvimento do protótipo

Serão descritos agora os passos realizados para implementar o programa que controla a nossa implementação do LFS. Para desenvolver nosso sistema experimental, optamos por adicionar as funções ao protótipo e testá-las paulatinamente, até que o protótipo se tornasse, após os testes finais, o produto acabado do projeto. Decidimos utilizar esta técnica pela facilidade de depuração do código que oferece. É muito mais produtivo e confiável testar e corrigir algumas funcionalidades por vez do que desenvolver todo o código do sistema de arquivos e então depurá-lo. Esta abordagem é possível pela facilidade de se agregar novas funções ao sistema de arquivos, proporcionada pelo protocolo do UserFS.

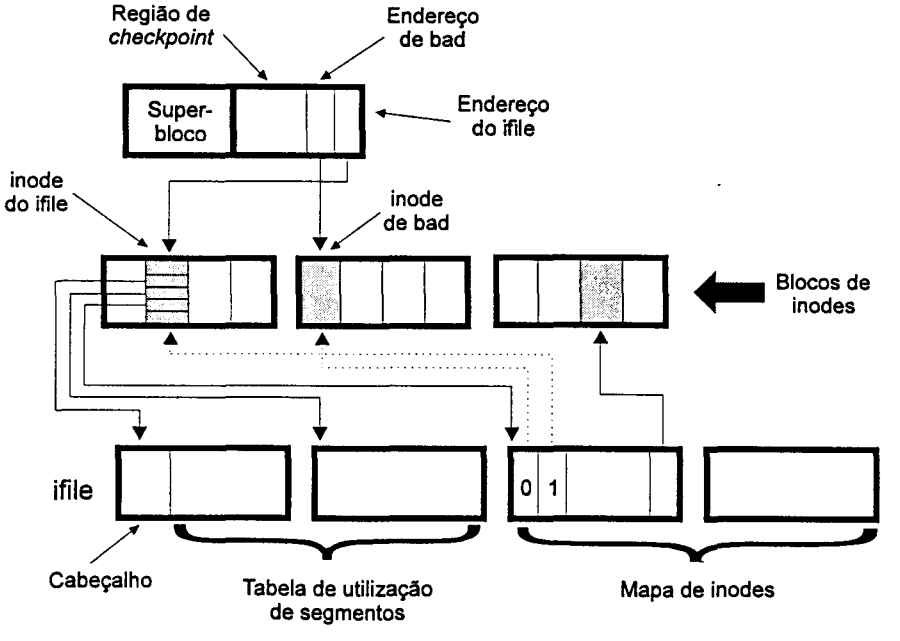


Figura 5.4: Como encontrar os inodes ifile e bad.

5.2.1 Experiências com o UserFS

Depois de um estudo aprofundado do LFS e do UserFS, partimos para os testes sobre esta ferramenta. Estes compreendem um sistema de arquivos experimental bastante simples, mas que elucidou vários aspectos da ferramenta e do VFS do LINUX que ainda estavam obscuros para nós. Este primeiro sistema de arquivos experimental que desenvolvemos, ao qual chamamos `dirfs`, funciona de forma análoga a um *link* para um diretório, tornando-o visível por outro nome e em outra posição dentro da árvore de diretórios. Embora pareça um tanto ingênua, esta aplicação explorou a maioria dos recursos do UserFS e de seu protocolo. Foi bastante válida também para verificar na prática a dinâmica das chamadas de sistema `mount` e `unmount`, bem como a realização de outras operações do VFS. Com o auxílio do aplicativo `gdb` (GNU Debugger) é fácil verificar quais as operações invocadas, por exemplo, quando o comando `ls` é executado. Para isso, basta verificar as mensagens recebidas do núcleo do UserFS.

Embora o UserFS traga uma biblioteca orientada a objetos para implementar os sistemas de arquivos experimentais, não a utilizamos neste projeto pois a nossa intenção é que o código gerado possa ser incorporado futuramente no núcleo do LINUX. Assim, a linguagem escolhida para todas as aplicações e programas de teste implementados foi C.

Quando o sistema de arquivos `dirfs` é montado, é passado o nome do diretório que será o seu ponto de montagem e também o nome do diretório que será manipulado pelo sistema de arquivos. Este segundo diretório já deve existir em algum sistema de arquivos montado. Por exemplo, o diretório `/home/user1/pub` poderia ser montado em `/home/ftp/pub` temporariamente, desde que o usuário que invocasse `muserfs` (seção 4.2.3) tivesse direitos de escrita sobre o diretório `/home/ftp/pub`. O comando para montar o diretório seria:

```
muserfs dirfs /home/ftp/pub -- /home/user1/pub
```

As informações que são colocadas depois de "`--`" são passadas pelo `muserfs` ao processo sem alterações. Após ser montado, o novo diretório "encobre" (*covers*) o antigo e os arquivos vistos por qualquer ferramenta, principalmente de FTP (*File Transfer Protocol*), são os contidos em `~user1/pub`. Um fato interessante no `dirfs` é que este não tem acesso direto a nenhum dispositivo de armazenamento, o que comprova a flexibilidade do UserFS para manipular outras estruturas, além de discos.

Para realizar a comunicação com o núcleo do UserFS, foram escritas duas primitivas chamadas `receive` e `reply`. A função `receive` recebe como parâmetro um dos descritores de arquivos do *pipe*, pelo qual lê os pacotes enviados pelo núcleo. Estes são "desempacotados" e então a função retorna, em parâmetros de saída, o cabeçalho da mensagem (em uma estrutura do tipo `up_preamble`) e o corpo da mensagem (como uma seqüência de bytes). A função `reply` recebe como parâmetros o outro descritor de arquivos do *pipe*, um cabeçalho do tipo `upp_repl` e uma seqüência de bytes, que corresponde à resposta da operação. Esta então monta um pacote *reply* e o envia pelo *pipe*.

O princípio de funcionamento do processo cliente do UserFS é muito simples. Este recebe as mensagens do tipo `up_request` do núcleo do UserFS, utilizando `receive`, identifica no cabeçalho qual a operação desejada e chama a sua função interna correspondente, passando a área de dados da mensagem como parâmetro. A função interna decodifica os dados utilizando as primitivas do protocolo e invoca a chamada de sistema correspondente à operação requisitada. Por exemplo, quando uma requisição `up_open` chega ao processo, `dirfs` identifica a operação e chama `dirfs_open`, que identifica o nome do arquivo a ser aberto e remete a operação ao VFS, invocando a chamada `open`. O resultado da operação é então "empacotado" e uma mensagem `up_reply` é montada e enviada para o UserFS.

Apenas as funções que não alteram o conteúdo de arquivos, como `lookup`, `open` e `read`, foram implementadas. Além disso, o sistema não guarda nenhum tipo de *cache* sobre as informação dos arquivos que foram recentemente requisitados. Assim, por exemplo, se duas operações `up_read` forem executadas consecutivamente em um mesmo arquivo, ainda que no mesmo *offset*, duas chamadas de sistema `read` serão invocadas.

5.2.2 mkulfs

A primeira aplicação que desenvolvemos para o novo sistema de arquivos foi um formatador, chamado **`mkulfs`**. Este insere em uma partição do disco as informações que descrevem características globais do sistema de arquivos e determinam seu estado inicial. Entre estas informações estão o superbloco, as regiões de *checkpoint* e o diretório raiz.

O formatador aceita diversos parâmetros passados em linha de comando. Estes são:

- **`b`**: tamanho do bloco. Atualmente pode estar entre 512 e 8192 bytes, deve ser múltiplo de 512 e potência de 2.
- **`c`**: não necessita de valor. Se este parâmetro for fornecido, a partição será verificada quanto a blocos ruins.
- **`d`**: nome do dispositivo que representa a partição a ser formatada. Por exemplo, `/dev/hda3`. Este argumento é exigido.
- **`f`**: número de fragmentos por bloco de dados. Atualmente pode variar entre 2 e 8 e deve ser potência de 2.
- **`i`**: se fornecido, indica que o sistema deve perguntar ao usuário antes de realizar efetivamente a formatação da partição.
- **`m`**: tamanho máximo do sistema de arquivos dado em kilobytes. Obviamente deve ser menor que o tamanho da partição.
- **`s`**: tamanho do segmento em kilobytes. O tamanho real do segmento será arredondado para o maior múltiplo do tamanho do bloco, que seja menor que o valor fornecido neste parâmetro. O tamanho mínimo permitido para o segmento é 128KB e o máximo 1024KB.

- `t`: intervalo máximo de tempo entre gravações do *checkpoint*. Atualmente encontra-se entre 5 e 3600 segundos.
- `u`: porcentagem de utilização mínima do segmento. Se um segmento gravado tiver menos que o estipulado neste parâmetro, este será reutilizado para gravar mais informações. A utilização mínima permitida atualmente é de 20%.

O programa verifica os parâmetros passados e também se os seus valores estão de acordo com os domínios permitidos para cada tipo de parâmetro. As opções `-c` e `-i` ainda não foram totalmente implementadas no protótipo. Por exemplo, ao receber a opção `-c`, o programa deveria verificar os blocos da partição, criar uma lista com os blocos danificados, caso houvessem, e gravá-la em disco para prevenir a utilização destes blocos. Geralmente esta lista é guardada em um arquivo especial invisível no sistema de arquivos. No `mkulfs`, o inode que guarda o arquivo especial é gravado em disco, mas a gravação das informações ainda não está completa.

Quando todos os dados necessários já foram extraídos das opções passadas pelo usuário e das características físicas da partição, o programa cria uma estrutura `ulfs_super_block` e uma estrutura `ulfs_checkpoint` em memória e as preenche com as informações sobre o novo sistema de arquivos. No final do processo de formatação, uma cópia do superbloco e a primeira região de *checkpoint* serão gravadas consecutivamente nesta ordem e ocuparão o primeiro bloco da partição. A segunda cópia do superbloco e a segunda região de *checkpoint* são colocadas no último bloco da partição. Isso é feito para facilitar o alinhamento de segmentos dentro da partição e para permitir a fácil localização destes metadados. Caso o tamanho máximo do sistema de arquivos, indicado no parâmetro `-m`, seja menor que o valor da partição, o sistema não criará uma segunda cópia do superbloco nem uma segunda região de *checkpoint*. Isso ocorre porque, na versão atual do `ulfs`, não há como o sistema localizar a posição desta segunda cópia se esta não estiver no último bloco da partição.

A principal razão pela qual a opção `-m` foi incluída foi para permitir que testes em sistemas com taxas de ocupação limite (com quase todos os segmentos ocupados) fossem realizados mais facilmente. Por exemplo, se a partição de testes possuir 500MB e se o sistema de arquivos ocupar toda a partição, devem ser gerados 500MB de dados e metadados para preencher todos os segmentos do sistema de arquivos antes que o *cleaner* seja invocado. Com a opção `-m` pode-se gerar sistemas de arquivos com um número pequeno de segmentos que são facilmente preenchidos.

Depois de criar o superbloco, o `mkulfs` cria os inodes especiais do sistema de arquivos. O primeiro é o inode que guarda a lista de blocos ruins da partição. O segundo é o inode do `ifile`, que guarda a tabela de utilização de segmentos e o mapa de inodes. O terceiro é o inode do diretório raiz do sistema de arquivos. Por fim, é criado o diretório `lost+found`, que será utilizado por ferramentas de recuperação do sistema de arquivos.

Todos estes inodes e os blocos de dados que fazem parte dos arquivos especiais são gravados no primeiro segmento fisicamente posterior ao superbloco. Este segmento

corresponde ao primeiro do *log* e sua organização é idêntica à dos demais segmentos que serão gravados.

Sempre que possível, as operações utilizadas no formatador para criar os inodes e gravar informações nos arquivos e diretórios são as mesmas utilizadas no funcionamento normal do sistema de arquivos. Assim, reutilizamos o mesmo código e economizamos no tamanho de todo o sistema. O mecanismo utilizado para ordenar o segmento e salvá-lo no disco também é o mesmo utilizado no programa `ulfs`.

Para testar o funcionamento do `mkulfs`, desenvolvemos uma outra aplicação simples que pudesse recuperar as informações do sistema de arquivos. Esta aplicação realiza a leitura do superbloco a partir do disco, o que assemelha-se ao processo de montagem do sistema, e fornece várias informações como número total de blocos, número de segmentos, último segmento escrito, entre outras. Novamente, a principal função utilizada é `ulfs_read_super`, usada também no protótipo `ulfs`. Como a função `ulfs_read_super` lê os inodes especiais do sistema de arquivos, o programa ainda fornece informações sobre o arquivo `ifile` e o diretório raiz, tais como a localização dos seus inodes, tamanho dos arquivos e a posição do primeiro bloco de dados de cada um.

5.3 O programa ulfs

Após os testes iniciais com o UserFS, com os quais adquirimos maior intimidade com o VFS do LINUX, partimos para a implementação do programa `ulfs`, que é o resultado final deste projeto. Atualmente, o programa `ulfs` possui cerca de 4.000 linhas de código em C e sua organização básica segue a mesma do `dirfs` (seção 5.2.1), desenvolvido anteriormente.

Os parâmetros recebidos por `ulfs` são os descritores de arquivos do *pipe* e o nome da partição que será manipulada. As primitivas de comunicação `receive` e `reply` de `dirfs` foram integralmente reaproveitadas e renomeadas para `ulfs_receive` e `ulfs_reply`.

Após receber e analisar os parâmetros, o processo entra em um laço que consiste em receber mensagens do UserFS, processá-las e enviar mensagens, quando necessário, em resposta aos pedidos feitos. Quando uma mensagem do tipo `up_request` é recebida, uma função interna correspondente à operação desejada é chamada. Por exemplo, quando uma mensagem `up_request` é recebida com uma operação `up_create` sendo solicitada, o processo `ulfs` invoca a função `create_r`, que utiliza uma função do UserFS para decodificar os parâmetros da chamada. Um trabalho adicional de `create_r` é decidir se o arquivo que está sendo criado é um arquivo comum ou um diretório. Isso é descoberto verificando-se o modo de criação do arquivo, que é um dos parâmetros passados. Se foi requisitado um arquivo comum, apenas a função `ulfs_inode_create` é chamada. Caso contrário, além de `ulfs_inode_create` ser chamada, são inseridas as entradas "." e ".." no diretório.

No modelo do UserFS, cada sistema de arquivos montado deve ser manipulado por um processo de usuário. Assim, o nosso sistema mantém apenas um superbloco em memória, deste o momento em que é montado até a hora em que é desmontado. Uma limitação do

sistema é que apenas uma requisição pode ser atendida por vez, ou seja, o programa não conta com mecanismos de *multi-threading*. Se por acaso alguma mensagem for enviada pelo núcleo enquanto o processo estiver respondendo a outra requisição, esta permanecerá no *pipe* até que o processo esteja pronto para ler uma nova mensagem.

As funções internas do sistema de arquivos passaram por sucessivos refinamentos e a cada estágio novas funcionalidades eram agregadas ao protótipo. Finalmente, a maior parte das operações do protocolo do UserFS e, em conseqüência, do VFS do LINUX, passaram a ser suportadas. As operações suportadas atualmente pelo sistema de arquivos estão na Tabela 5.1. Embora a semântica do UNIX não seja totalmente suportada pelo `ulfs`, especialmente em relação à atomicidade das operações de diretório, as operações deste protótipo cobrem os casos mais comuns de manipulação de arquivos e diretórios, incluindo criação, remoção, leitura, escrita e renomeação, entre outras.

Algumas das operações que não foram implementadas ou são pouco utilizadas ou não acrescentariam muito em termos de manipulação das estruturas do sistema de arquivos. Outras dizem respeito aos *links* simbólicos e necessitariam de algoritmos de análise de *paths* que tomariam um certo tempo de desenvolvimento. A operação `up_multreaddir` é uma otimização de `up_readdir` que retorna várias entradas de diretório de uma vez e pode ser substituída por esta última. A operação `up_permission` só necessita ser implementada se o sistema de arquivos exigir verificação de permissões para acesso a arquivos mais profunda que a semântica padrão do UNIX. Como o tempo de que dispúnhamos para desenvolver o protótipo era pequeno, decidimos deixar estas operações para um trabalho futuro.

O importante é que as operações implementadas cobrem a grande maioria das requisições feitas pelas aplicações. Prova disso é que vários testes foram realizados em um sistema de arquivos do tipo ULFS, utilizando editores de texto, compiladores e outras aplicações comuns, e estes exigiram apenas as operações que foram implementadas (seção 5.4).

O programa `ulfs` possui várias características interessantes das quais podemos destacar:

- O tamanho do bloco de dados pode variar de 512 bytes até 16KB, embora blocos de 4 ou 8KB sejam os mais indicados. O tamanho padrão é 4KB.
- A estrutura dos diretórios é bastante semelhante à utilizada no Ext2. As entradas possuem tamanhos variados, dependendo do comprimento do nome do arquivo (que pode ter até 255 caracteres). Quando uma entrada é apagada, esta é incorporada à entrada anterior no diretório e suas informações passam a ser desconsideradas. Posteriormente, o sistema pode requisitar este espaço para alocar uma nova entrada de diretório.
- A rotina de remoção de arquivos, `ulfs_inode_unlink`, cobre várias exceções. Estas incluem a remoção de arquivos diretório, que implica em saber se o diretório a ser removido está ou não vazio.

Protocolo do UserFS	ulfs
up_create[8]	ulfs_inode_create ulfs_inode_create + ulfs_dir_add
up_lookup	ulfs_inode_lookup
up_open	ulfs_file_open
up_close	ulfs_file_close
up_read	ulfs_file_read
up_write	ulfs_file_write
up_truncate	ulfs_inode_truncate
up_fsync	ulfs_fsync
up_readdir	ulfs_file_readdir
up_link	ulfs_inode_link
up_unlink[8]	ulfs_inode_unlink ulfs_inode_rmdir
up_symlink	Não implementada.
Up_readlink	Não implementada.
up_followlink	Não implementada.
up_mount	ulfs_mount
up_umount	ulfs_umount
up_iread	ulfs_inode_get
up_iwrite	Não implementada.
up_iput	Não implementada.
up_statfs	ulfs_statfs
up_permission	Não implementada.
up_rename	ulfs_inode_rename
up_multreaddir	Não implementada.
up_notify_change	ulfs_inode_notify_change
up_inode_valid	Não implementada.

Tabela 5.1: Mapeamento entre as operações do UserFS e do ulfs.

[8] A operação invocada depende do tipo do inode requisitado: arquivo comum ou diretório.

- Suporte a arquivos com buracos (*holes*). Esta é uma característica interessante da semântica do UNIX, que permite que arquivos possuam regiões em que não existem blocos de dados físicos alocados. Uma leitura que inclua parte de um buraco deve retornar zeros nesta região. Uma escrita em um buraco deve causar a alocação de um ou mais blocos de dados novos para o arquivo, além da atualização de seu tamanho, caso este tenha crescido.
- O `ulfs` é capaz de responder à operações `up_truncate` para qualquer número de bytes a partir de zero e até o tamanho máximo de arquivos que o `ulfs` permite. O algoritmo libera qualquer bloco alocado para o arquivo que esteja além do tamanho estipulado na chamada e ajusta o último bloco que restou para o novo tamanho do arquivo.

Decidimos implementar arquivos com buracos por causa de um fato curioso ocorrido quando realizávamos testes para verificar a correção do protótipo. Resolvemos, em um destes testes, copiar e compilar o próprio código do protótipo em uma partição `ulfs`. Percebemos então que o compilador `gcc` utiliza esta característica do sistema de arquivos do UNIX, realizando sucessivas escritas em um arquivo temporário em posições posteriores ao seu final físico. Assim, para dar prosseguimento ao teste, implementamos mais esta funcionalidade. O teste foi um sucesso e o arquivo compilado pôde ser executado como um processo comum. A operação `ulfs_inode_truncate` também foi implementada devido a necessidades encontradas durante os testes com o protótipo.

O `ulfs` conta também com um processo de limpeza de segmentos. Com isso, o sistema fica totalmente funcional e pode ser utilizado continuamente, enquanto a ocupação real do disco não atingir o máximo. A principal peça do processo de limpeza é o *cleaner*, implementado como uma função dentro do `ulfs`. Esta decisão foi tomada em detrimento da idéia inicial deste projeto de implementar o *cleaner* como um processo separado, ao estilo do BSD LFS [SEL93a]. O motivo principal para isso é que separar o *cleaner* exigiria que algum mecanismo de comunicação fosse implementado entre este e o processo que manipula a partição. Esta comunicação é necessária pois o sistema de arquivos, em certas ocasiões, precisa avisar ao *cleaner* que alguns segmentos necessitam ser limpos. O *cleaner*, por sua vez, necessita de serviços que só o código do sistema de arquivos pode oferecer, como o mapeamento de blocos lógicos dos arquivos para os endereços físicos dos blocos no disco. No BSD LFS isso é resolvido pela implementação de três novas chamadas de sistemas (seção 3.2.5). No nosso caso, por outro lado, seria necessário utilizar algum outro mecanismo, como IPC (*Inter Process Communication*) ou modificar o núcleo e o protocolo do UserFS para que as novas chamadas fossem suportadas. Isso complicaria o desenvolvimento do *cleaner*, resultando em um atraso na conclusão do trabalho.

Em sistemas reais, a existência do *cleaner* no núcleo do sistema operacional, como é o caso do Sprite LFS, diminui a modularidade do sistema de arquivos e impede que novas políticas de limpeza sejam testadas e utilizadas. Porém, em nosso caso, um sistema de arquivos que esteja sendo utilizado pode ser facilmente "desmontado" e um novo processo,

com um *cleaner* otimizado, pode ser compilado e testado sobre a partição. Assim, o impacto da inclusão do *cleaner* no programa `ulfs` foi irrelevante em termos de facilidade de uso.

Para deixar mais clara a operação do `ulfs` e algumas de suas funcionalidades, serão explicados agora alguns detalhes que consideramos mais importantes na sua execução. São abordados o processo de "montagem" e "desmontagem", a operação normal, a limpeza de segmentos e a recuperação de falhas.

5.3.1 Chamadas mount e unmount

O processo de "montagem" de um sistema de arquivos ULFS inicia-se quando um usuário chama a ferramenta `muserfs`. Esta dispara o processo `ulfs`, que fica esperando por uma mensagem `up_mount` vinda, via *pipe*, do núcleo do UserFS.

Quando uma requisição `up_mount` chega ao `ulfs` a função `mount_r` é chamada. Esta verifica se o dispositivo indicado como parâmetro pode ser aberto pelo processo para leitura e escrita e, após abri-lo, chama `ulfs_mount`. Esta função em especial chama `ulfs_read_super` que, além de ler o superbloco, descobre e carrega também a região de *checkpoint* mais recente. Depois disso, `ulfs_mount` realiza algumas inicializações nas estruturas em memória e lê os inodes especiais (`bad`, `ifile` e `root`), seguindo as informações do *checkpoint* e do próprio `ifile`. Caso o número de segmentos limpos esteja abaixo de um mínimo estipulado o mecanismo de limpeza de segmentos também é invocado em `ulfs_mount`.

O último passo de `mount_r` é obter o inode raiz do superbloco e devolver o seu número como *handle* para o UserFS. O processo de montagem do sistema de arquivos continua com o recebimento de uma requisição `up_iread` do UserFS, com o *handle* do inode raiz como parâmetro, que o `ulfs` responde com os atributos necessários deste inode. Depois disso o processo recebe e responde as mensagens *enquire* e está apto a responder aos demais *requests* feitos pelo núcleo.

A requisição `up_umount` causa a chamada da função `umount_r`, que por sua vez chama `ulfs_umount`. Esta força o sistema a realizar um *checkpoint*. Assim, todas as estruturas alteradas que estão apenas em memória são salvas para o disco. Quando este passo termina, as estruturas em memória que formam o sistema de arquivo são desalocadas. Por fim, `umount_r` fecha o dispositivo e o processo termina.

5.3.2 Operação normal

Após serem carregados na operação `mount`, o superbloco e a região de *checkpoint* permanecem em memória até que o processo receba uma requisição `up_umount`. No caso de blocos de dados, inodes e arquivos, estes são criados e desalocados dinamicamente.

Toda vez que um bloco de dados é requisitado, o programa verifica se este já foi trazido para a *cache*, procurando-o na tabela de *hashing* de blocos e no segmento corrente. Se não for encontrado, então é alocado um novo bloco vazio que será preenchido com dados vindos do disco.

Analogamente, quando algum inode é requisitado, o sistema verifica se este já está na tabela de *hashing* de inodes. Em caso negativo, o sistema verifica se existe algum inode na lista de livres. Se não existir, este aloca uma nova estrutura `ulfs_inode` e a preenche com os dados do disco.

O UserFS não utiliza descritores de arquivos ou números de inodes para comunicar-se com os processos de usuários. Ao invés disso, este utiliza um *handle* fornecido pelo processo de usuário e individual para cada inode requisitado. Como no ULFS o número do inode é um identificador exclusivo para cada inode, o *handle* passado ao UserFS é o próprio número do inode.

Quando o segmento corrente se enche ou quando um intervalo de tempo máximo é atingido, é invocada a rotina de ordenação, que determina a posição de cada bloco no segmento.

No momento da gravação do segmento, pode ocorrer que um ou mais inodes sujos, ou seja, que foram alterados e que necessitam ser gravados, estejam inconsistentes devido a operações de E/S que estejam em andamento sobre estes. Este estado é indicado em `ulfs_inode` por um *flag* `busy` e previne que inodes inconsistentes sejam gravados no *log*. Imagine, por exemplo, que uma operação `write` esteja sendo realizada sobre um arquivo. Considere que esta obriga o arquivo a expandir-se para mais um bloco. O primeiro passo do sistema, então, será alocar um novo bloco do segmento corrente para o inode do arquivo. Só depois os novos dados são copiados para o bloco recentemente alocado. Assim, se o segmento corrente tiver que ser descarregado (porque o período entre *checkpoints* espirou-se ou o segmento está cheio) antes que os dados sejam copiados para o bloco e se o inode for salvo, o arquivo estará inconsistente, pois o bloco fará parte do arquivo mas não conterá dados válidos.

Desta forma, o `ulfs` não grava inodes que tenham o *flag* `busy` com valor verdadeiro. Este método é eficiente para manter a consistência dos inodes mas, se uma precaução não for tomada, o sistema corre o risco de não ter mais espaço no segmento corrente para novos blocos. Por exemplo, imagine que logo após uma descarga do segmento corrente, um arquivo seja criado e diversas operações `write` sejam efetuadas sobre este. O arquivo pode crescer muito e ocupar mais de um segmento. Quando o último bloco do segmento corrente for alocado para o arquivo, o sistema deve descarregar o segmento. Mas, como o mecanismo de descarga está localizado nos níveis mais baixos das rotinas do sistema de arquivos, os dados que deveriam estar salvos no último bloco certamente não foram gravados e o inode possuirá o *flag* `busy` verdadeiro e não será gravado. Se os blocos do inode também permanecessem no segmento, não haverá mais espaço para outros blocos sujos, do mesmo arquivo ou de outro, e o sistema ficaria paralisado.

Para prevenir que isso aconteça, o ULFS adota a política de gravar os blocos sujos dos arquivos, mesmo que seus inodes não tenham sido salvos. Os inodes poderão ser salvos nas próximas descargas do sistema, sem comprometer a integridade dos dados. Outra política do ULFS é alterar a entrada do mapa de inodes para um certo inode apenas quando este for salvo

no *log*. Assim, no caso de uma ocorrência de falha entre a gravação dos blocos e do respectivo inode, o sistema considerará a antiga cópia do inode existente no *log*, onde estes novos blocos não existiam e, talvez, nem o inode existia.

Uma vez que os blocos são ordenados, são atribuídos endereços de blocos lógicos do sistema de arquivos, levando-se em consideração o segmento no qual estes serão gravados. O conteúdo de cada um destes blocos, que estão espalhados pela memória principal, é então copiado para um *buffer* contínuo, de onde vão para o disco. O segmento corrente então é limpo e preparado para uma nova série de operações de E/S. Os blocos que são gravados são marcados como limpos e são transferidos para a *cache* de blocos. Os inodes gravados também são marcados como limpos.

Se o segmento gravado contiver menos dados que o determinado no campo `min_seg_util` do superbloco, o mesmo segmento será reutilizado para gravar mais dados. Caso contrário, o próximo segmento é escolhido pela sua ordem no disco. O melhor caso seria o segmento imediatamente posterior ao que foi gravado. Caso este não esteja limpo, o processo continua procurando os segmentos à frente, até encontrar um limpo. Isto é feito para otimizar o posicionamento da cabeça de leitura/gravação, evitando grandes saltos no disco. Esta política funciona em especial quando existem muitas solicitações de escrita em um curto espaço de tempo.

5.3.3 Limpeza de segmentos

O *cleaner* do `ulfs` utiliza a política de custo-benefício desenvolvida no Sprite LFS (seção 3.1.4). Devido à simplicidade da *cache* de blocos de dados do protótipo, os blocos não possuem tempo de última modificação e a idade do segmento é dada pela hora da sua gravação no disco. Ainda não realizamos testes suficientes com o *cleaner* para saber se esta estratégia consegue a distribuição dicotômica de segmentos: alguns dos segmentos devem possuir taxas de utilização bastante baixas e os demais devem estar quase cheios, para que o processo de limpeza seja compensador em termos de desempenho.

O processo de limpeza é iniciado toda vez que um número mínimos de segmentos limpos é atingido. A função `ulfs_clean` é então invocada para escolher um segmento a ser limpo. Após o segmento ser escolhido, seguindo a política de custo-benefício, `ulfs_clean` chama a função `ulfs_seg_clean`, que efetivamente analisa os blocos de sumário do segmento, copia os blocos de dados ainda válidos e atualiza a tabela de utilização de segmentos.

À medida que vai copiando os dados e metadados válidos para a *cache*, o sistema marca-os como sujos, para que sejam gravados no disco novamente. Para assegurar que estes dados e metadados sejam realmente salvos, logo após terminar de verificar todo o segmento, o *cleaner* força a chamada de `ulfs_segment_flush`, causando um *checkpoint* do sistema de arquivos.

A função `ulfs_clean` é invocada até que um número máximo de segmentos limpos seja atingido ou até que não haja mais segmentos a serem limpos. O processo de limpeza

também pára quando, em alguma das iterações, o *cleaner* não consegue liberar mais nenhum segmento. Isso acontece quando a taxa de utilização do sistema está muito alta e a maioria dos dados dos segmentos ainda é válida. Assim, é utilizado um novo segmento limpo inteiro apenas para guardar os blocos de dados do segmento sendo limpo. Se o processo não fosse abortado, haveria o risco do sistema entrar em um laço infinito ou ficar completamente sem espaço livre. O processo também não tenta limpar exaustivamente todas as informações inválidas do sistema de arquivos pois isso poderia demorar um bom tempo e as aplicações teriam que esperar até que a limpeza terminasse. Como uma otimização, o *cleaner* também é chamado toda vez que o sistema de arquivos é montado, no intuito de diminuir a contenção com as chamadas normais das aplicações.

5.3.4 Recuperação de falhas

A recuperação de falhas no LFS depende especialmente das informações contidas nos sumários de segmentos e dos critérios de gravação de metadados e regiões de *checkpoint*. No caso do `ulfs`, as regiões de *checkpoint* são gravadas em intervalos de tempo regulares, determinados na hora de criação do sistema de arquivos. Seu sumário de segmento possui as informações necessárias para reconstruir qualquer estrutura do sistema de arquivos, de forma a deixá-lo em um estado consistente.

Caso ocorra uma queda do processo `ulfs`, possivelmente algumas informações da *cache* serão perdidas e um ou mais segmentos no final do *log* possuirão dados inconsistentes. Porém, quando o sistema é montado, o processo procura pela região de *checkpoint* que possui a marca de tempo mais recente e então recupera o `ifile` e as demais estruturas através das informações desta região. Assim, sempre haverá um estado consistente que pode ser recuperado do disco, como ficou comprovado nos testes (seção 5.4.1).

Por outro lado, caso uma falha no processo provoque alguma mudança nos dados do *log* anteriores ao último *checkpoint* gravado, não haverá mais consistência no sistema de arquivos. A recuperação das falhas neste caso depende de um programa especial, equivalente ao `fsck` do FFS. Este programa não era uma prioridade do projeto e resolvemos deixar a sua implementação para um trabalho futuro.

5.3.5 Características não implementadas

O sistema de arquivos do UNIX possui uma semântica complexa, pois prevê diversas exceções e exige comportamentos diferentes das operações de E/S sobre cada tipo de arquivo. Por exemplo, o fato de dispositivos, *pipes*, *sockets* e outras estruturas do sistema operacional (como o "pseudosistema" `/proc`, que fornece informações sobre a memória e os processos abertos) compartilharem a mesma abstração, obriga os modelos e as implementações dos

sistemas de arquivos a cobrirem as peculiaridades de cada objeto suportado. Um reflexo disso é a existência da operação `ioctl` na interface do VFS, que realiza operações específicas para cada um destes tipos de objetos. Além disso, os arquivos no UNIX possuem diversos atributos, tais como três marcas de tempo distintas e direitos de acesso, que devem ser mantidos corretamente. Assim, cobrir toda a semântica do UNIX não é simples.

Desta forma, decidimos implementar as funções do nosso protótipo de forma que suportassem a maioria das requisições das aplicações, mas sem nos atermos a todas as exceções previstas. Assim, alguns dos algoritmos devem ser melhorados para cobrirem estas exceções.

Uma característica importante não implementada é a consistência das operações de diretórios e metadados. Outra funcionalidade não implementada é a criação de arquivos especiais, como *pipes* e *sockets*. Em especial, as operações sobre *links* simbólicos não são permitidas.

Algumas otimizações existentes em outras implementações, em especial do FFS, também não foram feitas. Estas incluem a leitura de várias entradas de diretório com apenas uma operação, realizada por `multreaddir`, uma *cache* de entradas de diretórios, a leitura antecipada de blocos,

Um ponto importante de um sistema de arquivos é a recuperação de falhas. Como já foi mencionado, não desenvolvemos nenhum mecanismo dentro do `ulfs`, e nenhuma ferramenta, que realize o processo de *roll-forward* do LFS para recuperar os dados dos segmentos gravados após o último *checkpoint.*

Embora fosse planejado o uso de fragmentos no `ulfs`, sua implementação tomaria muito tempo do nosso projeto. Assim, decidimos apenas adaptar as estruturas do LFS para que suportassem mais esta característica.

5.4 Testes realizados

Todas as implementações, compilações e testes foram realizados em um computador PC equipado com processador Pentium de 90MHz, 32MB de memória principal e um disco rígido Quantum FireBall 1080A de 1,0GB de capacidade. Duas partições de 128MB foram separadas para testes.

O sistema operacional utilizado foi o LINUX 2.0.0, distribuição Slackware 3.1.0, com um núcleo compilado especialmente para a máquina acima. O núcleo do UserFS não foi alterado e foi compilado como um módulo do LINUX.

Como utilizamos a prototipação incremental para realizar nossa implementação, cada nova funcionalidade que era incluída no processo era testada imediatamente. Os testes para verificar as operações do `ulfs` foram feitos com aplicativos e comandos comuns, tais como `cp`, `ls`, `jed` (pequeno editor de textos baseado no `emacs`), `less` (aplicativo similar ao `more`), entre outros. Um outro teste realizado foi a compilação do próprio código do `ulfs` em um sistema de arquivos ULFS. O executável foi gerado perfeitamente. Os testes de recuperação de falhas e de desempenho exigiram abordagens diferentes, que serão apresentadas agora.

5.4.1 Recuperação de falhas

Para testar a recuperação do sistema após a ocorrência de falhas, utilizamos dois métodos. Primeiro, interrompemos deliberadamente, através do comando `kill`, a execução do processo enquanto este executava operações de E/S requisitadas por uma aplicação. Este procedimento foi realizado diversas vezes com aplicações distintas e estados diferentes do sistema de arquivos. Depois, inserimos funções `exit`, das bibliotecas padrão do C, em pontos do código nos quais a interrupção causaria, com certeza, a perda de informações vitais ao sistema. Em ambos os casos, como seria de se esperar o sistema perdia as informações gravadas no *log* posteriormente ao último *checkpoint*. Mas este sempre foi capaz de encontrar o *checkpoint* anterior e restabelecer um estado consistente.

5.4.2 Desempenho

Descreveremos agora alguns testes comparativos realizados entre o `ulfs` e o Ext2. O objetivo deste trabalho nunca foi realizar uma implementação otimizada do LFS. Sabíamos desde o início que o fato do ULFS ser implementado em nível de usuário o deixaria sujeito ao escalonamento do sistema operacional, o que o impede de executar com a prioridade de sistema de arquivos comum. Outro fato que prejudica o seu desempenho é a sincronização com o UserFS. Como utilizam um *pipe* em memória para comunicação, tanto o processo como o núcleo da ferramenta estão sujeitos aos mecanismos *wait* e *wakeup* do VFS do LINUX.

Em nossa implementação, para poder acessar diretamente uma partição não formatada, utilizamos operações de *raw I/O*, ou seja, são feitas chamadas de sistema `open`, `read`, `write` e `close` diretamente sobre o arquivo que representa o dispositivo da partição. Estes arquivos especiais de dispositivos são apenas um ponto de entrada em um sistema de arquivos qualquer, no caso do LINUX o Ext2, para os *drivers* dos dispositivos. Desta forma, as requisições feitas nestes arquivos não passam pela *buffer cache*, indo direto para as rotinas de tratamento dos *drivers* [RUS96]. No nosso caso, as requisições são tratadas pelo *driver* de discos rígidos do padrão IDE, que realiza a operação requisitada de forma síncrona no dispositivo físico. Assim, as operações de *raw I/O* que realizamos são síncronas, o que pesa muito no desempenho de um sistema de arquivos.

Todos os testes que realizamos foram feitos com o sistema utilizando apenas a memória principal, ou seja, com as áreas de *swap* desativadas. Isso foi feito para que os resultados não fossem alterados pelo tempo gasto nas trocas de páginas entre o disco e a memória. Embora o sistema estivesse em modo multi-usuário, apenas um usuário estava utilizando-o durante os testes. Assim, além dos programas de testes – e do `ulfs`, no caso dos testes com o protótipo – apenas alguns *daemons* do sistema operacional estavam sendo executados. Os testes apresentados aqui concentram-se nas operações de escrita, pois estas são as que apresentam os resultados mais interessantes.

Teste de escrita na partição

Um dos testes realizados foi idealizado para averiguar o impacto do sincronismo das operações de *raw I/O*. Este, chamado `wtest`, segue o seguinte algoritmo:

1. Criar `n` estruturas com `m` bytes, as quais chamamos de blocos, em memória, encadeando-as em um lista.
2. Transferir um número `p` destas estruturas para um *buffer* em memória, chamado de segmento.
3. Gravar o segmento em disco logo após o segmento anteriormente gravado.
4. Repetir 2 e 3 até que os blocos em memória terminem.

Este teste imita, de forma simplificada, o funcionamento do `ulfs`. O que se pretendia era ter uma idéia do tempo gasto pelo `ulfs` para alocar blocos e depois transferí-los para o disco. Duas versões do programa foram utilizadas: uma que realizava uma chamada `fsync` após cada escrita de segmento e outra que não realizava a chamada. Vários tamanhos de blocos e segmentos também foram testados, mas o montante de memória alocada em cada bateria de testes era sempre o mesmo. Cada configuração do programa foi executada diversas vezes e as médias dos resultados dos testes estão na Figura 5.5 (versão sem `fsync`) e na Figura 5.6 (versão com `fsync`). Os dados apresentados correspondem à alocação de 16MB de dados, que são depois gravados no disco. Como pode ser visto, foram tomados três tempos diferentes para cada teste. Estes correspondem aos três contadores que o LINUX dispõe para cada processo [DUB94a]. O primeiro, `REAL`, é o tempo real transcorrido deste a iniciação dos contadores até a sua leitura. O segundo, `VIRTUAL`, é o tempo que o processo ocupou o processador, ou seja, o tempo gasto para executar as instruções do programa. O último, `PROF`, corresponde ao tempo gasto pelo sistema operacional trabalhando em favor do processo, ou seja, execução do processo, troca de contexto, chamadas de sistema realizadas pelo processo, entre outras coisas.

Como podemos ver na Figura 5.5, os tempos variam pouco quando alteramos os tamanhos do bloco. A maior diferença que se nota é o tempo virtual, ou seja, o tempo de processamento. Isso acontece pela diferença na quantidade de blocos gerados em cada caso. Como o montante de dados alocados é sempre o mesmo, 16MB, a quantidade de blocos depende essencialmente do tamanho do bloco, e quanto menos blocos alocados, mais rápido é o processamento. Outros tamanhos de segmentos também foram testados e os números variaram muito pouco dos que são apresentados.

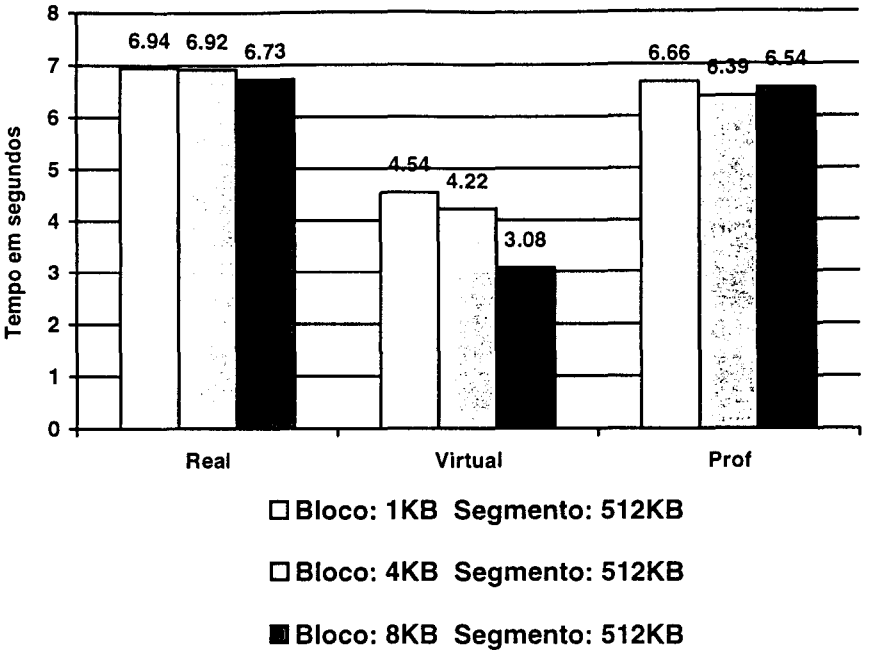


Figura 5.5: Resultados dos testes de escrita na partição sem fsync.

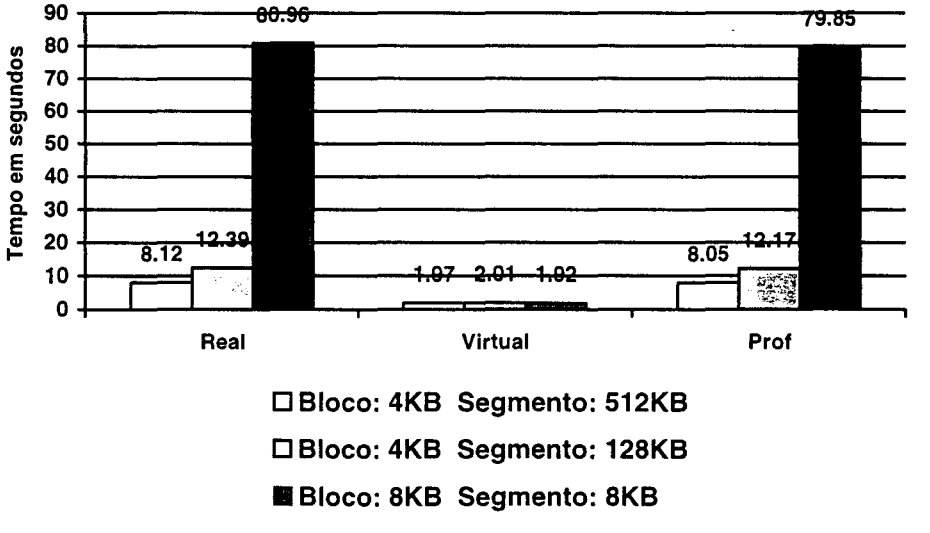


Figura 5.6: Resultados dos testes de escrita na partição com fsync.

Na Figura 5.6, os tempos são um pouco mais altos. Como já foi visto, as operações de *raw I/O* são síncronas e poderia se pensar que `fsync` não teria nenhum efeito, ou seja, não haveriam blocos sujos do dispositivo para serem salvos. Porém, o que `fsync` faz é atualizar em disco o inode do arquivo do dispositivo, que teve o atributo tamanho alterado após a chamada `write`. Como o inode está em um sistema de arquivos em outra partição no disco, isso força o disco a deslocar a cabeça de leitura/gravação de um lado para o outro, ocasionando este aumento no tempo de resposta.

A terceira coluna de cada tempo mostra o impacto que esta movimentação pode ocasionar. Esta coluna reflete um teste em que o segmento tinha exatamente o tamanho de um bloco, ou seja, a cada bloco gravado em disco era feita uma chamada `fsync`. Isso nos levou a crer que, se o sistema Ext2 fosse utilizado em seu modo síncrono, seu desempenho deveria cair substancialmente. Quando um sistema de arquivos Ext2, ou FFS, é montado no modo síncrono, qualquer mudança nos metadados (inodes, blocos de apontadores, arquivos de diretório, descritores de grupos de blocos e mapas de bits), são realizadas imediatamente no disco [CAR93]. Esta perda de desempenho foi comprovada com mais um teste que realizamos.

Teste de criação de arquivos

Na comparação entre o `ulfs` e o Ext2, criamos um teste de escrita exaustiva, no qual utilizamos um programa simples, chamado `crtest`, baseado no seguinte algoritmo:

1. Criar um arquivo.
2. Copiar o próprio nome dele 1000 vezes em um *buffer* em memória.
3. Gravar este *buffer* no arquivo com apenas uma chamada `write`.
4. Repetir os passos 1, 2 e 3 `n` vezes.

Repetimos os testes diversas vezes, variando o número de arquivos criados. A Figura 5.7 mostra as médias dos resultados obtidos para a criação de 100 arquivos e a Figura 5.8 traz a média dos resultados da criação de 1000 arquivos. Em ambas as figuras, os arquivos possuem 9KB cada. Os valores mais expressivos são os do contador `REAL`, pois os outros dois não possuem variações expressivas entre os três sistemas, uma vez que indicam o tempo gasto pela aplicação e não pelos sistemas de arquivos. O que se pode verificar é que o Ext2 assíncrono é realmente muito mais rápido que o `ulfs`, pois realiza a maior parte das operações em *cache*. Já o `ulfs` necessita realizar uma operação `write` após cada segmento, devido à organização da sua *cache*. Porém, quando comparado ao Ext2 síncrono, o `ulfs` ganha com grande vantagem. O tempo gasto pelo Ext2 síncrono para 1000 arquivos foi quase 140s, o que é 7 vezes mais lento que o `ulfs`. Como outros trabalhos já comprovaram, em *workloads* onde as operações sobre metadados predominam, o modelo do FFS pode ser até 10 vezes mais lento que o LFS.

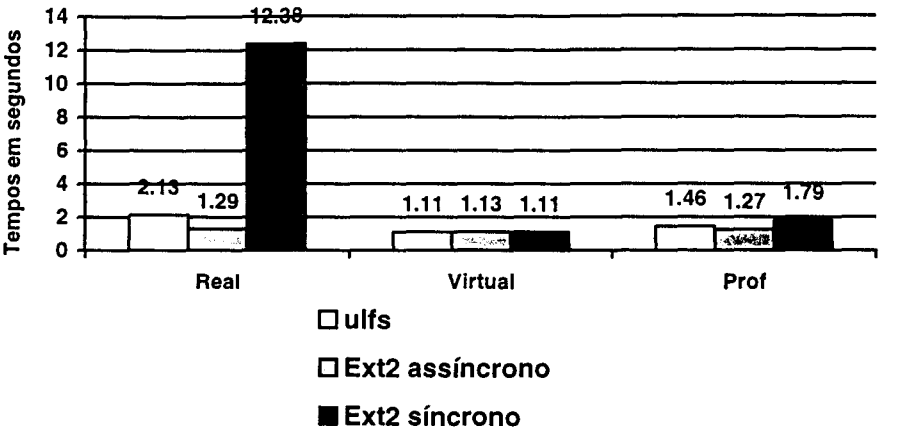


Figura 5.7: Resultados da gravação exaustiva de 100 arquivos.

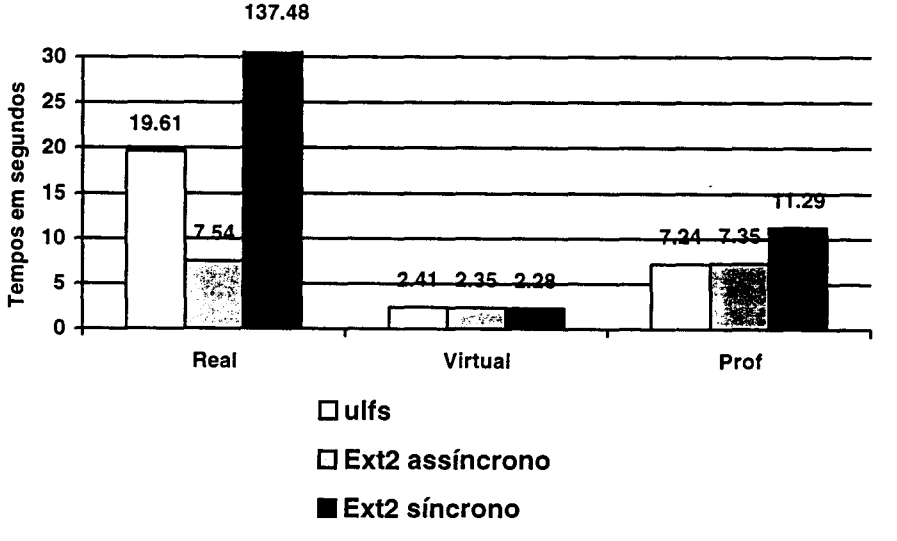


Figura 5.8: Resultados da gravação exaustiva de 1000 arquivos.

Os resultados mostrados são de testes realizados com as partições inteiramente vazias. Nos testes de rescrita, ou seja, quando os arquivos deviam ser apagados, recriados e escritos novamente, os resultados mudam um pouco. O Ext2 assíncrono ficou cerca de 40% mais rápido. Acreditamos que isso aconteça pois grande parte dos dados ainda estão em *cache* quando são rescritos, uma vez que as cópias dos dados gravados permanecem em memória até que o espaço ocupado por estas seja necessário. É importante observar que o tempo medido é o tempo de resposta do sistema de arquivos para a aplicação e não o tempo total gasto para gravar efetivamente os dados no disco, o que depende de políticas de descarga da *cache*. O `ulfs` foi cerca de 15% mais lento, pois não dispõem, por exemplo, de *cache* de entradas de diretório. Outra razão é que o `ulfs`, como está hoje, necessita descarregar o segmento sempre que este se enche. O Ext2 síncrono foi cerca de 8% mais rápido, também possivelmente devido à *cache*. O benefício da *cache* não foi tão grande quanto o do Ext2 assíncrono pois este último não necessita esperar pelas gravações.

Pelos resultados obtidos nos testes, concluímos que o ULFS tem grandes chances de se igualar ao Ext2 assíncrono, no que diz respeito à escrita de arquivos pequenos, se este contar com uma *cache* de dados mais bem desenvolvida e for inserido no núcleo do LINUX. Esta melhora também se aplica ao caso do Ext2 síncrono, onde o `ulfs`, mesmo sendo um protótipo, já é várias vezes mais rápido.

5.5 Conclusões

Neste projeto, estudamos um dos mais promissores modelos de sistemas de arquivos desenvolvidos nos últimos anos, o LFS. Este foi comparado com o FFS, modelo de sistema de arquivos que é o padrão *de facto* dos sistemas UNIX atuais. Comparamos também suas principais implementações, enfatizando suas diferenças no que diz respeito à organização física dos dados e às estruturas de cada uma. As características mais interessantes e os principais problemas que devem ser resolvidos neste modelo também foram apresentados.

Para contribuir com o desenvolvimento do LFS, implementamos uma versão deste no sistema operacional LINUX. Reunimos as melhores características de cada uma das duas principais implementações do LFS e propusemos algumas adaptações nas suas estruturas. As principais mudanças foram feitas no sumário de segmento, que em nosso protótipo é um híbrido entre a organização do Sprite LFS e do BSD LFS. Estas mudanças são decorrentes da inclusão de blocos fragmentados no modelo.

5.5.1 Avaliação da ferramenta UserFS

Vários meses de utilização do UserFS, levaram às seguintes impressões. De forma geral, o UserFS é uma ferramenta de grande utilidade para o desenvolvimento de sistemas de

arquivos experimentais, sem a qual seria difícil construir sistemas de arquivos a nível de usuário. A idéia de construir seu núcleo como um módulo do LINUX torna-o mais robusto, estável e menos susceptível aos processos do usuário. No nosso caso, após vários testes, inúmeras operações `mount` e `unmount` e mesmo com erros no nosso processo, o UserFS nunca deixou o subsistema de arquivos em um estado inconsistente ou causou a queda do sistema. Desta forma, a atenção no desenvolvimento do sistema de arquivos fica toda voltada para a manipulação da partição, sem que seja necessário se preocupar com a correção do código que se comunica com o VFS, que já foi testado com diversos outros sistemas experimentais.

Uma das metas deste projeto era manter a interface das operações do `ulfs` o mais próximo possível do VFS do LINUX. Porém, o protocolo do UserFS não segue fielmente as operações do VFS. Assim, tivemos que utilizar funções intermediárias entre o protocolo do UserFS e as funções do `ulfs` que realmente manipulam as estruturas do sistema de arquivos. Embora a maioria destas funções apenas "desempacote" os parâmetros da mensagem e "empacote" o resultado na área de dados da resposta, algumas delas realizam um tratamento maior antes de chamarem as funções finais. Por exemplo, algumas requisições do UserFS não diferenciam *handles* de inodes e de arquivos, mas a camada VFS possui parâmetros distintos para cada um destes objetos (respectivamente descritor de arquivos e número do inode). Assim, no caso de uma chamada `up_read`, o `ulfs` deve procurar primeiramente o arquivo que possui o *handle* passado e só então invocar a função `ulfs_file_read`. Se o UserFS implementasse um protocolo um pouco mais fiel ao VFS, seria mais fácil esta ligação entre o processo e o seu núcleo.

5.5.2 Dificuldades encontradas

Nos sistemas operacionais modernos, o bom desempenho de um sistema de arquivos depende tanto de seu projeto quanto da sua integração com os outros subsistemas, em especial com o VMS. Uma das maiores dificuldades que encontramos para implementar o ULFS foi desenvolver os mecanismos em memória que controlam a *cache* de dados e metadados. Caso o sistema fosse implementado diretamente no núcleo do sistema operacional, o resultado da implementação em relação ao desempenho, poderia ser bem melhor do que o conseguido.

A bibliografia sobre o LFS, como acontece em geral com a maioria dos textos técnicos, focaliza muito bem os aspectos teóricos do modelo do sistema de arquivos, inclusive com riqueza de resultados de *benchmarks*. Porém, diversos detalhes de implementação não são explicados e a dinâmica das estruturas auxiliares e seu relacionamento com o sistema operacional são omitidos para simplificar os textos. Assim, a fonte que possuíamos como referência para realizarmos a implementação eram os códigos das implementações anteriores, em especial do BSD LFS, do UserFS e do núcleo do LINUX, com seu subsistema de arquivos. Geralmente, a interpretação destes códigos não é tão simples como a leitura de um texto técnico e toma muito mais tempo. Além disso, os subsistemas de memória virtual do LINUX e do UNIX BSD não são compatíveis e tivemos que construir um novo mapeamento de

endereçamento de blocos na *buffer cache*, pois pretendíamos continuar a utilizar o esquema do LINUX.

O código de um sistema de arquivos geralmente é bastante intrincado e um pequeno erro em uma função pode determinar a perda de todos os dados de uma partição. Este problema é ainda mais crítico no LFS, onde as estruturas que contém os metadados estão quase todas espalhadas pelo *log* e a consistência dos dados está baseada no encadeamento destas estruturas e no mecanismo de *checkpoint*. Mesmo as implementações mais maduras do LFS, como a do Sprite e do BSD possuem *bugs*, inclusive conceituais. Assim, boa parte do último semestre do projeto foi gasta realizando-se correções no código do `ulfs`.

5.5.3 Futuras extensões

Como o programa `ulfs` é um protótipo, entendemos que este já tenha servido ao seu propósito e que o seu aprimoramento não traria benefícios ao trabalho para portar o LFS para o LINUX. Além disso, as características que lhe estão faltando podem ser implementadas diretamente no VFS do LINUX. Assim, a extensão natural do trabalho seria implementar uma versão do LFS no núcleo do sistema LINUX. Esta deveria possuir a maior parte das características do `ulfs`, mas com algumas correções e aprimoramentos ainda não realizados. Por exemplo, os algoritmos do `ulfs` não são otimizados nem preparados para concorrência. Desta forma, seria indispensável a adaptação destes aos mecanismos do VFS, tais como a *buffer cache* e o *daemon* `bdflush`.

Um grande problema do LFS é a integração com a *cache* de dados do sistema operacional. Nenhuma solução apresentada até agora cobre satisfatoriamente as exigências tanto do LFS como dos sistemas operacionais. O Sprite LFS utiliza demasiada memória para implementar suas estruturas e o BSD LFS adotou soluções deselegantes para integrar-se ao VMS do BSD 4.3. Assim, um trabalho futuro poderia estudar as reais exigências do LFS quanto à memória principal e modificar a *cache* do LINUX, de forma a permitir que tanto sistemas já implementados quanto o LFS compartilhassem um mesmo mecanismo. Este novo VMS poderia ser baseado em soluções mais modernas, adotadas em sistemas operacionais mais recentes como o BSD 4.4 e o SVR4 [VAH96].

Faltam também algumas ferramentas auxiliares, além do formatador `mkulfs`, que também poderia ser melhorado. Em especial, poderia ser implementado um programa recuperador do sistema de arquivos, que restaurasse o *log* em caso de danos causados à partição.

Além da implementação em si, o próprio LFS pode ser melhorado. Este é um modelo relativamente novo e com um bom potencial de desenvolvimento. Diversas características podem ser atacadas para que haja um aumento no desempenho e na segurança deste modelo. Vários parâmetros poderiam ser variados para realizar uma "sintonia" do sistema de arquivos. Estes incluem o tamanho ideal dos segmentos, dos blocos de dados, o número de fragmentos por bloco e o valor do intervalo entre *checkpoints*. Além disso, alguns algoritmos podem ser ligeiramente modificados para implementar novas políticas, como a ordenação dos blocos de

dados e metadados no segmento (o que pode influenciar as leituras seqüenciais de arquivos). Algoritmos que mudam o comportamento do sistema em tempo de execução também podem ajudar a melhorar o desempenho em diversas situações de uso [MAT97].

O processo de limpeza de segmentos também é um vasto campo onde experimentos poderiam ser realizados. Novas políticas de invocação do *cleaner*, que levam em conta a carga do sistema de arquivos e os períodos ociosos de tempo (seção 3.2.6) parecem ser a melhor solução para minimizar o impacto que a limpeza de segmentos provoca no sistema. Outros pontos a serem estudados são a quantidade de dados ou segmentos a serem limpos em cada invocação do *cleaner*, o uso de *cleaners* em paralelo e estratégias de compactação e reorganização dos arquivos no *log* (visando o posicionamento seqüencial dos seus blocos de dados).

Por fim, o ULFS ainda necessita de mais testes comparativos com outros sistemas de arquivos, em especial com aqueles que utilizam a técnica de *journaling* [KAZ90, VER96, SIL98], os quais possuem princípios de funcionamento bastante semelhantes com os deste modelo e que vêm tornando-se os mais utilizados nos sistemas operacionais mais modernos .

Apêndice A

Principais estruturas de dados do ulfs

A.1 Super bloco

A.1.1 ulfs_static_info

```
/* Super bloco em disco. */
struct ulfs_static_info {
   char id[32];                  /* string de ident. do s. a. */
   char version[32];             /* versao do sistema de arquivos */
   unsigned int number;          /* numero do super bloco */
   unsigned int block_size;      /* tamanho do bloco do s. arq. em bytes */
   unsigned int inode_size;      /* tamanho do inode em bytes */
   unsigned int seg_size;        /* tamanho do segmento em blocos */
   unsigned int fragments;       /* num. de fragmentos por bloco */
   unsigned int addr_per_blk;    /* num. de apontadores por bloco */
   unsigned int sut_per_blk;     /* num. de 'ulfs_sut_entry' por bloco */
   unsigned int imap_per_blk;    /* num. de 'ulfs_imap_entry' por bloco */
   unsigned int inode_per_blk;   /* num. de inodes por bloco */
   unsigned int sent_per_blk;    /* numero de ulfs_summ_entry por bloco */
   unsigned int min_seg_util;    /* taxa minima de reutil. de um seg. */
   unsigned int reserv1;         /* reservado */
   unsigned long part_size;      /* tam. da particao em blocos de 512b */
   unsigned long fs_size;        /* tam. do sist. arq. em blocos logicos */
   unsigned long ctime;          /* hora da criacao */
   unsigned long segments;       /* num. total de segmentos */
   unsigned long max_inodes;     /* num. maximo de inodes */
   unsigned long flush_interval; /* intervalo maximo entre gravacoes */
};
```

A.1.2 ulfs_checkpoint

```
/* Regiao de checkpoint em disco e em memoria. */
struct ulfs_checkpoint {
   unsigned long current_segment; /* segmento atual */
   unsigned long timestamp;       /* utima atualizacao do checkpoint */
   unsigned long ifile_addr;      /* endereco do inode do ifile */
   unsigned long bad_addr;        /* endereco do inode do bad file */
   unsigned long first_free_ino;  /* primeiro inode da lista de livres */
   unsigned long inode_count;     /* total de inodes no sist. de arq. */
   unsigned long lst_summblk;     /* ultimo bloco de sumario gravado */
   unsigned int current_offset;   /* 1o. bloco do seg. parcial corrente*/
   unsigned int ifile_offset;     /* posicao do inode no bloco de inodes */
   unsigned int bad_off;          /* posicao do inode no bloco de inodes */
   unsigned int status;           /* estado atual do superbloco */
   unsigned int reserv1;          /* reservado */
};
```

A.1.3 ulfs_super_block

```
/* Superbloco em memoria. */
struct ulfs_super_block {
   struct ulfs_static_info static_sb;     /* informacoes estaticas */
   struct ulfs_checkpoint check_reg;      /* regiao de checkpoint */
   struct ulfs_segment cur_seg;           /* segmento corrente */
   struct ulfs_inode *inode_table[INO_HASHT]; /* t. de hashing de inodes */
   struct ulfs_block *block_table[BLK_HASHT]; /* t. de hashing de blocos */
   struct ulfs_file *files;               /* lista de arquivos abertos */
   struct ulfs_inode *bad;                /* inode dos bad blocks */
   struct ulfs_inode *ifile;              /* inode do ifile */
   struct ulfs_inode *root;               /* inode do diretorio raiz */
   struct {
      struct ulfs_ifile_head *h;          /* cabecalho do ifile */
      struct ulfs_block *blk;             /* bloco do cabecalho do ifile */
   } if_head;
   unsigned long sb2blk;                  /* pos. do 2o super-b. em disco */
   long file_cnt;                         /* num. de arquivos abertos. */
   long num_inodes;                       /* num. de inodes */
   unsigned long clean_blocks;            /* num. de blc. limpos na cache */
   int busy;                              /* estado do superbloco */
   int dev_fd;                            /* fd. da particao montada */
   int check_turn;                        /* que regiao de checkpoint sera
                                             gravada */
};
```

A.2 Arquivos

A.2.1 ulfs_disk_inode

```
/* Inode em disco. */
struct ulfs_disk_inode {
   unsigned int mode;                  /* modo do arquivo */
   unsigned int uid;                   /* id do usuario */
   unsigned int gid;                   /* id do grupo do usuario */
   unsigned int links;                 /* links que apontam p/ o inode */
   unsigned int fragments;             /* mapa de bits do ultimo bloco */
   unsigned long number;               /* numero do inode */
   unsigned long size;                 /* tamanho em bytes*/
   unsigned long ctime;                /* hora de criacao */
   unsigned long mtime;                /* hora da ultima modificacao */
   /* Lembre-se que 'atime' vai no imap */
   unsigned long blocks;               /* numero de blocos */
   unsigned long flags;                /* flags apenas por compatib. */
   unsigned long pointers[INO_BLK_PTRS]; /* apontadores p/ blocos de dados
                                            e blocos de apontadores */

};
```

A.2.2 ulfs_inode

```
/* Inode em memoria. */
struct ulfs_inode {
   struct ulfs_disk_inode *dsk_ino;    /* informacoes do disco */
   struct ulfs_super_block *sb;        /* apontador para o superbloco */
   struct ulfs_block *block;           /* bloco de inodes a que pertence */
   unsigned long atime;                /* hora do ultimo acesso */
   int offset;                         /* desloc. no bloco de inodes */
   unsigned int version;               /* versao do inode */
   int busy;                           /* esta' sendo alterado? */
   int status;                         /* Dirty / Clean / Free */
   char in_summ;                       /* Ja esta no sumario? */
   struct ulfs_inode *hash_prev;       /* anterior na tab. de hashing */
   struct ulfs_inode *hash_next;       /* proximo na tab. de hashing */
};
```

A.2.3 ulfs_file

```
/* Arquivo aberto. */
struct ulfs_file {
   unsigned short f_mode;              /* modo de abertura */
   long long f_pos;                    /* posicao do apontador */
   unsigned short f_flags;
```

```
   unsigned short f_count;
   struct ulfs_file *f_next, *f_prev; /* encadeamento na lista */
   struct ulfs_inode *f_inode;        /* inode ao qual pertence */
};
```

A.2.4 ulfs_dir_entry

```
/* Cada entrada de um arquivo diretorio */
struct ulfs_dir_entry {
   int ent_size;                  /* tamanho total da entrada */
   unsigned long inode;           /* numero do inode */
   int name_len;                  /* tamanho do nome do arquivo */
   char name[ULFS_MAX_NAMELEN];   /* nome do arquivo */
};
```

A.3 Bloco de dados

A.3.1 ulfs_blk_type

```
/* Tipos de blocos encontrados em cache. */
enum ulfs_blk_type {
   Summary,
   Inodeblk,
   Data,
   Bad
};
```

A.3.2 ulfs_block

```
/* Bloco do sistema de arquivos em cache. */
struct ulfs_block {
   enum ulfs_blk_type type;       /* tipo do bloco */
   int status;                    /* Dirty / Clean */
   int num_inodes;                /* usado apenas para o tipo Inodeblk */
   unsigned long address;         /* endereco do bloco */
   char *buf;                     /* dados */
   struct ulfs_super_block *sb;   /* apontador para o superbloco */
   struct ulfs_block *hash_prev;  /* anterior na tab. de hashing */
   struct ulfs_block *hash_next;  /* proximo na tab. de hashing */
};
```

A.4 Segmento corrente

A.4.1 ulfs_segment

```
/* Segmento corrente que guarda os blocos sujos da cache */
struct ulfs_segment {
   long number;                    /* numero do segmento sendo escrito */
   unsigned long min_addr;         /* primeiro endereco do segmento */
   unsigned long cblk_cnt;         /* contador de blocos limpos */
   long offset;                    /* inicio do segmento parcial */
   long num_dblks;                 /* numero de bloco sujos */
   long num_dinodes;               /* numero de inodes sujos */
   int freeblk_lst;                /* lista de blocos livres */
   int inoblk_lst;                 /* lista de blocos de inodes */
   int summblk_lst;                /* lista de blocos de sumario */
   struct ulfs_seg_entry *blks;    /* vetor de blocos do segmento */
   struct {
      struct ulfs_summ_header *head;  /* cabecalho do sumario */
      struct ulfs_block *blk;      /* bloco do sumario */
      int cnt;                     /* num. entradas no sumario */
   } summ;                         /* controle do sumario sendo utilizado */
   int *seg_lst;                   /* lista de blocos apos a ordenacao */
   unsigned char *buffer;          /* buffer para escrita em disco */
   };
```

A.4.2 ulfs_seg_entry

```
/* Cada entrada do vetor de blocos do segmento corrente. */
struct ulfs_seg_entry {
   struct ulfs_block *blk;      /* apontador para o bloco do s. a. */
   struct own {
      unsigned long inode;      /* numero do inode */
      long pos;                 /* posicao logica no arquivo */
      unsigned int version;     /* versao no inode */
   } *owners;                   /* inodes que tem fragmentos no bloco */
   int next;                    /* proximo da lista */
   char fragmented;             /* esta fragmentado? */
};
```

A.5 Sumário de segmento

A.5.1 ulfs_summ_header

```
/* Cabecalho do sumario de segmento. */
struct ulfs_summ_header {
   unsigned long sum_check;        /* checksum do sumario */
   unsigned long data_check;       /* checksum dos blocos de dados*/
   unsigned long ctime;            /* hora de criacao do sumario */
   unsigned long entries;          /* numero de entradas */
   unsigned long prev;             /* sumario anterior no log */
   unsigned long next;             /* proximo sumario no log */
};
```

A.5.2 ulfs_summ_entry

```
/* Cada descritor do sumario de segmento. */
struct ulfs_summ_entry {
   unsigned long inode;       /* inode ao qual pertence ou ULFS_FRAG_BLK */
   long pos;                  /* posicao do bloco no inode */
   unsigned int version;      /* numero de versao do inode */
};
```

A.6 Ifile

A.6.1 ulfs_ifile_head

```
/* Informacoes existentes no ifile para o cleaner. */
struct ulfs_ifile_head {
   unsigned long clean_segments;     /* num. de segmentos limpos */
   unsigned long free_blocks;        /* numero de blocos livres no s.a. */
   unsigned long current_seg;        /* num. do segmento corrente */
   unsigned int head_size;           /* tamanho do cabecalho em bytes */
   unsigned long fst_imap_blk;       /* primeiro bloco do mapa de inodes */
   unsigned long inodes;             /* numero total de inodes */
   unsigned long fst_free_ino;       /* cabeca da lista de inodes livres */
};
```

A.6.2 ulfs_sut_entry

```
/* Cada linha da tabela de utilizacao dos segmentos, no ifile. */
struct ulfs_sut_entry {
   unsigned long live_blks;      /* numero de blocos vivos no segmento */
   unsigned long modif_time;     /* hora de gravacao do segmento */
   unsigned int flags;           /* Clear, Used, Current, Beeing clean */
};
```

A.6.3 ulfs_imap_entry

```
/* Cada linha do mapa de inodes, no ifile. */
struct ulfs_imap_entry {
   int offset;                     /* posicao do inode no bloco */
   unsigned long address;          /* bloco no qual o inode se encontra */
   unsigned long version;          /* versao do inode */
   unsigned long atime;            /* hora do ultimo acesso */
   unsigned long next_free;        /* proximo na lista de livres */
};
```

Referências Bibliográficas

[BLA95] Blackwell, T.; Harris, J.; Seltzer, M.; "*Heuristic Cleaning Algorithms in Log-Structured File Systems*"; Relatório técnico, Harvard University; 1995.

[BUR92] Burrows, M.; Jerian, C.; Lampson, B.; Mann, T.; "*On-line Data Compression in a Log-structured File System*"; Proceedings of the Fifth Symposium on Programming Languages and Operating Systems, out. 1992, pág. 2 a 9.

[CAR93] Card, R.; Ts'o, T.; Tweedie, S.; "*Design and Implementation of the Second Extended Filesystem*"; Proceedings of the First Dutch International Symposium on Linux, 1993.

[CAS98] Cassidy, C.; "*Understanding the Performance of Quantum Solid State Disks*"; White Paper, Quantum Corporation; 1998. Disponível em: http://www.quantum.com/src/whitepapers/upqsd

[CHE94] Chen, P. M.; Lee, E. K.; Gibson, G. A.; Katz, R. H.; Patterson, D. A.; "*RAID: High-Performance, Reliable Secondary Storage*"; ACM - Computing Surveys, vol. 26, nº 2, jun. 1994, pág. 145 a 185.

[COO98] Cook, C.; "*Linux Log-structured Filesystem Project*". Disponível em: http://collective.cpoint.net/prof/lfs

[DEC96] Digital Equipment Corporation; "*Spiralog File System for Open VMS Alpha*"; Open VMS InfoCenter - Information Sheet, 1996. Disponível em: http://www.openvms.digital.com/cgi-bin/textit.exe/openvms/announcements/spiralog/technical.html

[DUB94a] Dubeau, L. D.; "*Linux 1.0 Syscalls*"; The Linux Documentation Project; 1994.

[DUB94b] Dubeau, L. D.; "*Analysis of the Ext2fs structure*"; The Linux Documentation Project; 1994.

[FIT93] Fitzhardinge, J.; "*UserFS - Filesystems Implemented as User Processes*"; Documentação distribuída com a ferramenta, Softway Pty. Ltd., 1993.

[GOL95] Goldt, S. et all; "*The Linux Programmers' Guide*"; The Linux Documentation Project; 1995.

[GRE96] Green, R. J.; Baird, A. C.; Davies, J. C.; "*Designing a Fast On-line Backup System for a Log-structured File System*"; Digital Technical Journal; vol. 8, nº 2, 1996.

[HAR95] Hartman, J. H.; Ousterhout, J. K.; "*The Zebra Striped Network File System*"; ACM - Transactions on Computer Systems, vol. 13, nº 3, ago. 1995, pág. 274 a 310.

[HEN94] Hennessy, J. L.; Patterson, D. A.; "*Computer Organization and Design*"; Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, 1994.

[JMK95] Johnson, M. K.; "*The Linux Kernel Hackers' Guide*"; The Linux Documentation Project; 1995.

[JJE96] Johnson, J. E.; Laing, W. A.; "*Overview of the Spiralog File System*"; Digital Technical Journal; vol. 8, nº 2, 1996.

[KAR86] Karels M.; McKusick, M. K.; "*Toward a Compatible Filesystem Interface*"; Conference of the European UNIX Users´ Group; 1986.

[KAZ90] Kaz, M. et al; "DECorum File System Architectural Overview"; Proceedings of the USENIX 1990 Summer Conference, Anaheim, CA, 1990, pág 151 a 164.

[KLE86] Kleiman, S. R.; "*Vnodes: An Architecture for Multiple File System Types in Sun UNIX*"; Proceedings of the USENIX 1986 Summer Conference; pág 238 a 247.

[LDP98] "*The Linux Documentation Project Home Page*"; Disponível em: http://sunsite.unc.edu/LDP

[MAT97] Matthews, J. N. et al; "*Improving the Performance of Log-Structured File Systems with Adaptive Methods*"; Sixteenth ACM Symposion on Operating Systems Principles, Saint Malo, França, out., 1997.

[MCK84] McKusick, M. K.; Joy, W. N.; Leffler, S.; Fabry, R. S.; "*A Fast File System for UNIX*"; ACM - Transactions on Computer Systems, vol. 2, nº 3, ago. 1994, pág. 181 a 197.

[MCV91] McVoy, L. W.; Kleiman, S. R.; "*Extent-like Performance from a UNIX File System*"; Proceedings of the USENIX 1991 Winter Conference, Dallas, TX, 1991.

[OUS85] Ousterhout, J. K. et al; "*A Trace-Driven Analysis of the UNIX 4.2 BSD File System*"; Proceedings of the Tenth ACM Sysposium on Operating Systems, Orca Island, Washington, USA, dez., 1985.

[OUS88] Ousterhout, J. K.; Douglis, F.; "*Beating the I/O Bottleneck: A Case for Log-Structured File Systems*"; Technical Report, University of California at Berkeley, CA, USA, 1988.

[OUS90] Ousterhout, J. K.; "*Why Aren't Operating Systems Getting Faster As Fast as Hardware*"; USENIX Summer Conference, Anaheim, CA, USA, jun, 1990.

[QUA98a] Quantum Corporation, 1998. Disponível em: http://www.quantum.com/src/whitepapers/mtbf/qntmtbf4.htm

[QUA98b] "*Using Quantum Solid State Disks in RAID Systems*"; White Paper, Quantum Corporation; 1998. Disponível em: http://www.quantum.com/src/whitepapers/raid

[ROS92a] Rosenblum, M.; Ousterhout, J.; "*The Design and Implementation of a Log-Structured File System*"; ACM - Transactions on Computer Systems, vol 10, nº 1, fev. 1992, pág. 26 a 52.

[ROS92b] Rosenblum, M; "*The Design and Implementation of a Log-structured File System*"; Tese de Doutorado, University of California at Berkeley, jun. 1992.

[RUS96] Rusling D. A.; "*The Linux Kernel*", versão 0.8-2; The Linux Documentation Project. Disponível em: http://sunsite.unc.edu/LDP/tlk/tlk.html

[SAN87] Sandberg, R. "*The Sun Network File System*"; Sun Microsystems, Inc; Mountain View, CA, USA, 1987.

[SEA98] Seagate Corporation; Disponível em: http://www.seagate.com/disc/cuda/cuda91p18launch.shtml

[SEL92] Seltzer, M.; "*File System Performance and Transaction Support*"; Dissertação de Tese de Doutorado, University of California at Berkeley, USA, 1992.

[SEL93a] Seltzer, M.; Bostic, K.; McKusick, M. K.; Staelin, C.; "*An Implementation of a Log-structured File System for UNIX*", Proceedings of the USENIX 1993 Winter Conference, San Diego, CA, USA, jan. 1993, pág. 307 a 326.

[SEL93b] Seltzer, M.; "*Transaction Support in a Log-Structured File System*"; Proceedings of the 1993 International Conference on Data Engineering, abr. 1993.

[SEL95] Seltzer, M.; Smith, K. A.; Balakrishnan, H.; Chang, J.; McMains, S.; Padmanabhan, V.; "*File System Logging Versus Clustering: A Performance Comparison*"; Proceedings of the USENIX 1995 Winter Conference, 1995.

[SIL98] SiliconGraphics Incorporation; "*XFS File System Data Sheet*", SiliconGraphics Inc., Mountain View, CA, USA; 1998. Disponível em: http:www_europe.sgi.com/Products/hadware/challenge/XFS/XFS.html

[TAN87] Tanenbaum, A.; "*Operating Systems: Design and Implementation*"; Prentice-Hall Inc., Englewood Cliffs, NJ, USA; 1987.

[VAH96] Vahalia, U.; "*UNIX Internals: The New Frontiers*"; Prentice-Hall Inc, Englewood Cliffs, NJ, USA; 1996.

[VER96] "*VERITAS File System*"; VERITAS Software Corporation; Mountain View, CA, USA; 1996. Disponível em: http://www.veritas.com/htmldocs/products/fileSys/fileSys.html

[WHI96] Whitaker, C.; Bayley, J. S.; Widdowson, R. D. W.; "*Design of the Server for the Spiralog File System*"; Digital Technical Journal; vol. 8, nº 2, 1996.

[WIL96] Wilkes, J.; Golding, R.; Staelin, C.; Sullivan, T.; "*The HP AutoRAID Hierarchical Storage System*"; ACM - Transaction on Computer Systems, vol. 14, nº 1, fev. 1996; pág. 108 a 136.

[WIR95] Wirzenius, L.; "*Linux System Administrator's Guide*"; The Linux Documentation Project; 1995.